

Game Development in C++

Paul Pedriana
Senior Software Engineer
Maxis, Electronic Arts
March 15, 1999

Contents

CONTENTS	2
OUR GOAL.....	4
REMEMBER THIS.....	5
C++ CHARACTERS	6
ANDY ACADEMIA	6
SAMMY C.....	6
PETER PORTABILITY.....	6
PATTY PLUG-IN	7
NANCY NIH.....	7
GARY GOOD GUY	7
TEN WAYS C++ IS NOT ANSI C.....	8
C++ COMPILERS	10
C++ FEATURES FOR GAME PROGRAMMING.....	13
CLASSES AND INHERITANCE	13
PUBLIC/PRIVATE/PROTECTED	21
INLINING.....	23
POINTERS TO MEMBER FUNCTIONS.....	24
OPERATOR OVERLOADING.....	26
CUSTOM ALLOCATORS.....	29
STL.....	33
STRING CLASS.....	35
AUTO-PTR	37
EXCEPTION HANDLING.....	38
RUN-TIME TYPE IDENTIFICATION.....	41
STREAM IO	43
TEMPLATES.....	48
EXAMPLE OF COOL C++ CLASS.....	49
HIGH PERFORMANCE C++.....	53
MIXING C, C++, AND ASSEMBLY.....	54
C++ NAME MANGLING.....	54
CALLING C FROM C++.....	54
CALLING C++ FROM C.....	55
CALLING ASSEMBLY FROM C++.....	55
CALLING C++ FROM ASSEMBLY.....	55
INLINE ASSEMBLY WITH C++	55

C++ PORTABILITY.....	58
LOCALIZATION WITH C++.....	60
WIN32 LOCALIZATION SUPPORT	61
C/C++ LOCALE SUPPORT	63
CUSTOM LOCALE SUPPORT	64
GAME FRAMEWORKS IN C++.....	69
PLATFORM-SPECIFIC CODE.....	73
FACTORIES.....	74
EXAMPLES.....	75
FRAMEWORK CONCLUSIONS.....	79
PLUG-IN SYSTEMS IN C++.....	80
MICROSOFT COM.....	82
PLUG-INS AT MAXIS.....	84
THE DISTRIBUTION DISK.....	86
THE MINI FRAMEWORK.....	87
NO 3D?.....	87
BASIC GRAPHICS.....	93
MORE GRAPHICS.....	95
COMPRESSION.....	97
RANDOM NUMBERS.....	99
FILE IO.....	101
THREADS	102
C++ RESOURCES	103
BOOKS.....	103
PERIODICALS	105
INTERNET RESOURCES.....	107
GURUS	108

Our Goal

Our goal is to finish the day coming away with confidence in doing a real-world game project developed with C++.

We're going to dissect C++ features – the good and the bad. We're going to discuss the major types of C++ programmer personalities. We're going to discuss a number of real-world C++ development issues and your best approach to dealing with them. This class is as much a sharing of years of personal experience as it is a presentation of facts and figures. The material should be best-appreciated by programmers and technical directors. A lot of what we're going to talk about relates to programming design and policies, so technical directors and programming managers have as much to learn here as programmers. Some of the example code we present here is specific to Microsoft Visual C++. Nevertheless, these examples are usually easily extended to other compilers. There simply is not enough space or time to provide examples for all available compilers.

Remember This

If you remember anything from today, remember these things:

- C++ can be every bit as efficient as C, sometimes more so. If anybody tries to tell you otherwise, then they don't know what they're talking about. If you or they do think they know what they're talking about, then come to me.
- Make your C++ classes as independent of each other as possible.
- Don't do things a certain way just because you saw somebody else do it that way, regardless of whether that person is an expert.
- Design code and classes that are simple and straightforward, rather than obfuscated or academic.
- Don't try to design the world all at once. Take small but well-thought baby steps. There's something about C++ that makes programmers start getting these grand visions of super architectures.
- In writing a C++ framework, make sure the application code is fully independent of the framework. The framework code itself is independent and has no idea what application it is running under.
- In writing a C++ framework, make sure the resource loading/management system is visible only at the application level and not visible to the framework.
- Programmers hate jumping through a lot of hoops to do simple things. Don't force this on them with the classes you create.
- Just because C++ offers `private` and `protected` data doesn't mean you have to use it all the time. Ditto for other features.

C++ Characters

C++ Characters are common programming personality types that you'll find. For the most part, the examples given are of the more extreme variety. But remember that they are composite characters – in reality, most programmers are somewhere in between and not as extreme. Nevertheless, you'll no doubt recognize them...

Andy Academia

- Insists on preparing for things that will never happen. For example, he may insist on making even basic low-level structures have protected data and virtual accessors, claiming, "What if we need to this class to be protected some day? We're burning our bridges now by letting people have direct access to the data. It's going to cause maintenance problems."
- Language Lawyer – Knows obscure details of the language standard and likes to use these details regularly.
- Wants to make all data for all classes private and all functions virtual. We saw this in the example above.
- Complains about how this or that compiler isn't compliant.
- Story: We had a guy like that. His questionable forward declaration of a class member template class worked on VC++ 5.01 and 5.03, but not 5.00 and 5.02.

Sammy C

- The reverse of Andy Academia.
- Dislikes C++ and "militant" C++ programmers like Andy Academia.
- Insists that you can do everything in C that you could do in C++, and it's easier to read, since it is clear what is happening, as opposed to C++ with its hidden mechanisms.
- Story: We had a guy like this too. He had this old DOS file editor that accepted 8.3 files only and got really mad when we started switching to using long file names. Here we finally had the holy grail of using a PC, long filenames, and he hated it.

Peter Portability

- Always concerned with code portability and readability.
- Often wants to insist that all programmers use "lint" to test for compatibility.
- Wants all programmers on a project to wrap all lines after 80 characters.
- Wants all filenames stored in 8.3, despite the fact that you'll likely never actually write for DOS ever again.
- Story: There was once this guy who once complained that I was not putting "void" in the argument declaration of a C++ class function that took no arguments. His reasoning was that if we ever re-implement the code in C then this would cause porting problems.

Patty Plug-In

- She's really hooked on the plug-in paradigm.
- Something of an idealist. Has cool plug-in ideas, but they're sometimes more work that is worth it. Also, the realities of development sometimes get in the way, such as schedules do.
- Story: When we started working on our current C++ game application framework, we had a windowing system, graphic system, framework manager, and 2D and 3D rasterization systems. Our Patty Plug-In wanted to make these all plug-innable. Ideally, in theory, we could simply replace the 2D rasterization system with a new one and none of the other systems would be affected. This is a noble idea, but is not as easy as it may sound. This worked well for entities that used the rasterizer as a black box, such as the windowing system, but not for entities that needed to really work intimately with the internals of the rasterizer, such as other parts of the rasterizer itself. In the end, we settled for a middle-of-the-road solution, which we'll see later when we talk about plug-in architectures in C++.

Nancy NIH

- This is actually not a C++ character but more of a general programming character.
- Insists that she could fix a bug faster by re-writing the entire algorithm from scratch than by trying to figure out what the original author meant to do.
- Also, likes to write code from scratch when there is public domain code that does the same thing already available.
- Story: I once wrote a text editor for our previous interface library. We were updating the library to be compatible with the current interface library. He had just come on board, fresh out of school but with some independent programming experience under his belt. Anyway, he wanted to rewrite the text editor from scratch – saying it would take longer to figure out my code than he could implement it himself. I told him that it took me 5 days to get that text editor working and debugged. If he sat down and patiently studied my code, he could have it entirely understood in one day. He complained and I said he had no choice. It turns out that he had it completely figured out in about 3 hours and had it ported by the next morning.

There's a lesson here: In the school programming world, you are punished for copying other people's work, whereas in the real programming world, you should be *rewarded* for copying other people's work!

Gary Good Guy

- This is actually our model of a good C++ programmer.
- Doesn't always need to rewrite code (unlike Nancy-NIH). Takes the time to understand other's code and learns to work with it. Copies code whenever possible.
- Uses C++ for what it was meant: a tool that makes some things easier or more powerful.
- Is not interested in programming politics.
- Tends to focus on the problem being solved, not the details of how it gets solved.
- Story: We've got people like this kind of guy too.

Ten Ways C++ is not ANSI C

Here we present a top ten list of reasons C++ is not C. Even though C++ compilers normally compile C code without problems (as if C++ was merely a superset of C), there are some times when there are problems, largely due to C++'s more strict type checking. This top ten list is adapted from a section from *Dave's Book of Top Ten Lists for Windows Programming*, by Dave Edson. What we do here is boil it down to just the essentials in an easy-to-digest list. I hope this helps you understand a little better how C++ works and why some of these changes in design were made.

1	void*	<p>In C a <code>void*</code> can be assigned to and from any other pointer without a cast. For example, this is legal in C:</p> <pre>void* pNothing; char* pData = pNothing;</pre> <p>This is illegal when compiled by a C++ compiler. In C++ <code>void*</code> is treated much like it is a parent class of all pointers. Thus, you assign any pointer to <code>void*</code> but not the other way.</p> <p>Why then, does this code compile in C++:</p> <pre>char* pData = NULL;</pre> <p>given that in C, <code>NULL</code> is defined as <code>(void*)(0)</code>?</p> <p>The answer is that in C++, <code>NULL</code> is usually defined as <code>(0)</code>.</p>
2	Pointer Conversion	<p>In C, this code is legal:</p> <pre>Car* pCar; //No relation between Car and Rat. Rat* pRat = pCar;</pre> <p>This is clearly illegal when compiled by a C++ compiler, not without <code>reinterpret_cast<></code>, at least.</p>
3	Main is Not a Normal Function in C++	<p>In C, this is legal code:</p> <pre>int main(int argc, char** argv){ main(argc, argv); //call main() from main() }</pre> <p>This is illegal in C++. In C, <code>main()</code> is nothing but another function. In theory, you can call it from another function or take the address of it. This is not true in C++. You can neither call it nor take the address of it.</p>
4	Type Name Spaces	<p>In C, this is legal code:</p> <pre>struct A{ int a; }; struct B{ int b; }; typedef struct A B;</pre> <p>This is illegal in C++. The basic reason is that the first <code>A</code> is treated as if it is in a different declaration space than the second <code>A</code>.</p>

5	Enumerations	<p>In C, this is legal code:</p> <pre>enum C{ c1, c2, c3, }; enum C c = c3; c++; c*=2;</pre> <p>This is illegal in C++. However, you <i>can</i> make it legal by defining <code>operator++()</code> and <code>operator*=()</code> for the enum. You would do it just like for any regular C++ class.</p>
6	Scoping	<p>In C, this is legal code:</p> <pre>struct A{ struct B{ int x; }; }; B b;</pre> <p>This is illegal in C++. In C, the compiler treats the above statement as if it was written like this:</p> <pre>struct B{ int x; }; struct A{ B internalB; };</pre>
7	Default Global Constant Linkage	<p>In C, this is legal code:</p> <pre>const int global = 1; //Source File 1 extern const int global; //Source File 2 global = 2; //Source File 2</pre> <p>In C++, this is illegal; it generates a link error. This is because when you declare something <code>const</code> at file scope in C++, it is also implied to be <code>static</code>. This is normally an advantage, because it reduces the global space and allows the compiler to implement the constant inline in code.</p>
8	Function Prototypes	<p>In C, this is legal code and declares and generates a total of one function:</p> <pre>int DoSomething(); int DoSomething(int x, int y){ return x+y; }</pre> <p>In C++, the above code declares two functions. If this is all you wrote, then you would get a linker error because <code>int DoSomething()</code> was not found.</p>
9	Character String Initialization	<p>In C, this is legal code:</p> <pre>char str[4] = "abcd";</pre> <p>This is illegal in C++, because "abcd" requires a size of at least five chars.</p>
10	Type of Character Literals	<p>In C, <code>sizeof('A') == sizeof(int)</code>. In C++, <code>sizeof('A') == sizeof(char)</code>. Enough said.</p>

C++ Compilers

We're going to do a small survey of the currently available C++ compilers for various game development platforms. I am not recommending any of these compilers in particular; you have to make that decision for yourself. In fact, there's a good chance that you've already standardized on some compiler and this section doesn't mean a whole lot to you. For that reason, this section is brief.

Comparing and judging compilers and development systems in general is tough. Debates on the topic are religious and have sapped megabytes of many news servers' disk drives. We're going to try not to start any big arguments here. I'm sure you have some kind of horror story about some compiler or another. I personally have no love or hate for any of the currently available compilers, including Microsoft's compiler. And I've at one time or another used all the compilers discussed below, though I have more experience with some than others. I did have some pretty bad experiences with the old Symantec C++ v6 for Mac...

Here is a list of the primary C++ development systems available currently. There are others, but they are used less often:

Compiler	Supported Platforms	URL	ANSI/ISO C++ Complaint?	Current Version as of 3/99
Microsoft	Windows	www.microsoft.com	No	6.0.2
PowerSoft (formerly Watcom)	Windows, DOS	www.sybase.com		Power++ 2.1 Watcom C++ 11.0
Inprise/Borland	Windows, DOS	www.inprise.com www.borland.com	Yes	C++ Builder 4.0*
Intel (Requires Microsoft C++)	Windows	developer.intel.com/vtune/icl/index.htm		3.0
MetroWerks Code Warrior	Windows, Mac, PlayStation, Sun Solaris, others	www.metrowerks.com	Yes	4.1*
Symantec	Windows	www.symantec.com/scpp/fs_scpp72_95.html	No	7.5
GNU G++	Windows, DOS, Mac, PlayStation, others	http://www.gnu.ai.mit.edu/software/gcc/gcc.html also, newsgroups: gnu.c++	No	2.8.1*
DJGPP (GNU port)	DOS	www.delorie.com/djgpp/	No	2.0.2*

*Free downloads of demonstration versions are currently available for these development systems

Game programmers are perhaps most interested in the optimization capabilities of the compiler, and less interested in client/server development wizards. In this area, the Intel compiler shines. Here are some miscellaneous highlights regarding these compilers:

- The Microsoft compiler optimizes well, second only to Intel (and MetroWerks?), but it sometimes misgenerates code (C and C++) and sometimes fails to compile, giving "internal exception" errors. This is a fact of life with this compiler. However, it may require some expertise to tell the difference between misgenerated code and mistyped code. Nevertheless, one clue that it *may* be misgenerating code is that the code works with optimizations turned off, but not when on. However, even the unoptimized version may be misgenerated. There are a number of workarounds, which usually involve writing the same thing a different way or moving the offending function to a different place. Also, try disabling optimizations for just the function that is being misgenerated, like this:

```
#pragma optimize( "", off)    //turn off optimizations
<function here>
#pragma optimize( "", on)     //turn on previous optimizations
```

Simple rule for upgrading to any new version of Microsoft compiler: Wait until the version has at least a ".02" after it (as in v6.02).

- The source code to the GNU compiler is public domain, subject to the GNU usage and redistribution conventions. This means that with some work, you can get the GNU compiler to work on new platforms. Just because this compiler is GNU-ware and not commercial doesn't mean it isn't good. In fact, this is a very good compiler.
- The venerable Watcom compiler has been bought by PowerSoft, and unfortunately for us, they are more interested in corporate client-server development than high performance game development. Time will tell us where PowerSoft takes this compiler.
- Intel C++ is only available as a plug-in to Microsoft VC++ Professional. It used to be available under Borland C++, but that doesn't seem to be the case any more. Perhaps we can see other compiler vendors such as Borland and Metrowerks use Intel as a plug-in in the future.
- MetroWerks CodeWarrior is currently the only major C++ development system for Macintosh, but it is a good system which supports a number of other platforms as well, including the PlayStation. It probably deserves the award for best cross-platform support. The interface is very Mac-centric, which is refreshing if you long for that Macintosh look and feel.
- The latest MetroWerks compiler claims to be faster than Microsoft for a number of industry-standard benchmarks.

- Borland C++ has always been known for being the most up-to-date with the C++ standard and having the fastest compiler, but its code optimization has generally lagged slightly behind the Intel's and Microsoft's compilers. This may not matter much to programmers who take the time to hand-optimize the key bottleneck sections of their code. We have found that with big projects, compile and link time makes a significant difference during development, especially during milestone crunches. The new incremental compiler and linker in BC++ helps out here too.
- Microsoft Visual C++ claims to have incremental linking, but this is actually not completely true. VC++ can do incremental linking only if you've made changes to the main application project. Imagine you have your game source code divided up into four sub-projects:
 - * Simulation
 - * Graphics
 - * UI
 - * Main executable
 If, for example, you make changes to any of the code in the UI sub-project, no incremental link can happen. Unfortunately, the situation where you want incremental linking is for big projects that are divided into sub-projects; this is exactly the case where VC++ incremental linking fails.
- VC++ v6 has a nifty feature that allows you to rebuild your app while it is still running and continue from your last breakpoint. Initial tests of this are promising.

C++ Features for Game Programming

In this section, we see why you would want to use C++ for game programming. I presume that you are already familiar with the basic features and syntax of C++, so I won't waste your time or trees describing the fundamental features of the language. *However*, I do want to talk about how these features can be best used for game programming. This section is at the center of this entire manual; all future sections derive from and build upon this section.

Classes and Inheritance

Inheritance is a great vehicle for some aspects of game programming. Luckily, the original designers of C++ had efficiency as one of their primary objects. Some proponents of pure object oriented languages like SmallTalk deride C++ for its static typing (that is, all types and subclasses are fixed at compile time). A C++ object is of one type, and while inheritance can extend that type through a base pointer, the scope of the type is still limited. Well, this "limitation" is what allows C++ to be so efficient. Subclasses of the primary class can be accessed just as fast as through the primary class. Even overloaded virtual functions are very fast.

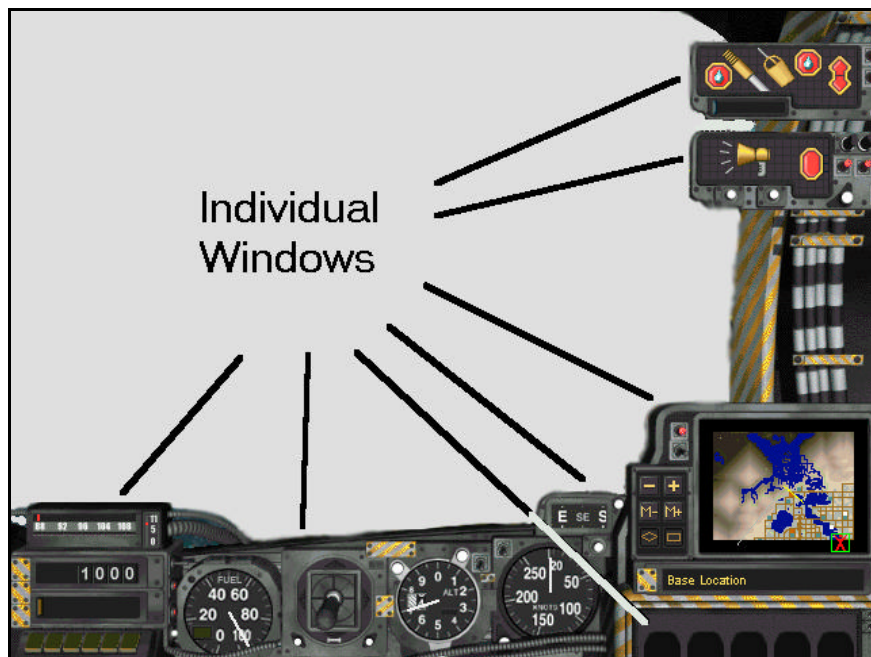
We're going to examine a few applications of inheritance in game programming. Most likely, you've already pondered the cool applications of C++ classes and inheritance. I'm not here to teach you how inheritance works; you probably understand that pretty well already. I am here to convince you that inheritance can be used in virtually any part of your game, from the user interface to the renderer.

Custom Windowing System

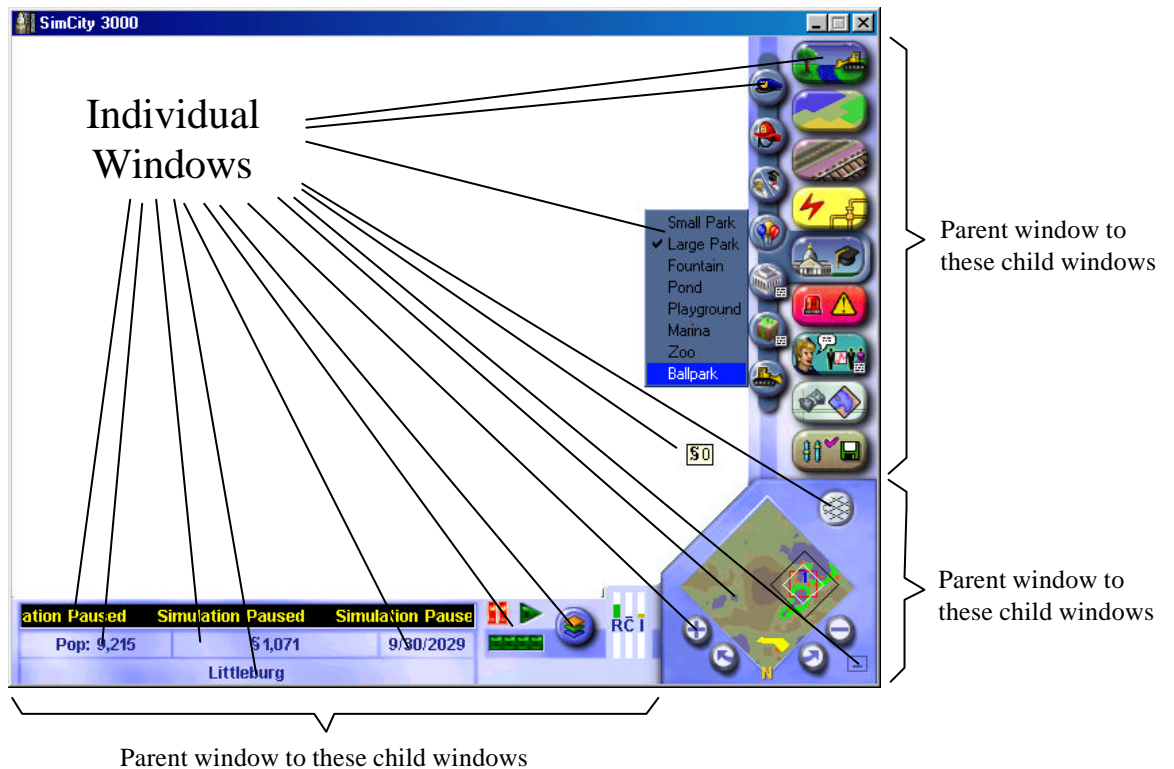
Let's examine the implementation of a game windowing system implemented in C++. The windowing system we're going to examine is not an imaginary one, but the one we use at Maxis. We wrote this system for a number of reasons:

- HWNDs (Microsoft Windows' window types) are virtually incompatible with DirectX. Even with "owner-draw" windows, you can't implement a game with Windows' windows.
- MFC is virtually incompatible with DirectX, and simply doesn't lend much help to a DirectX app anyway.
- We wanted a *portable* interface library—one that would let us use the same code for multiple platforms.

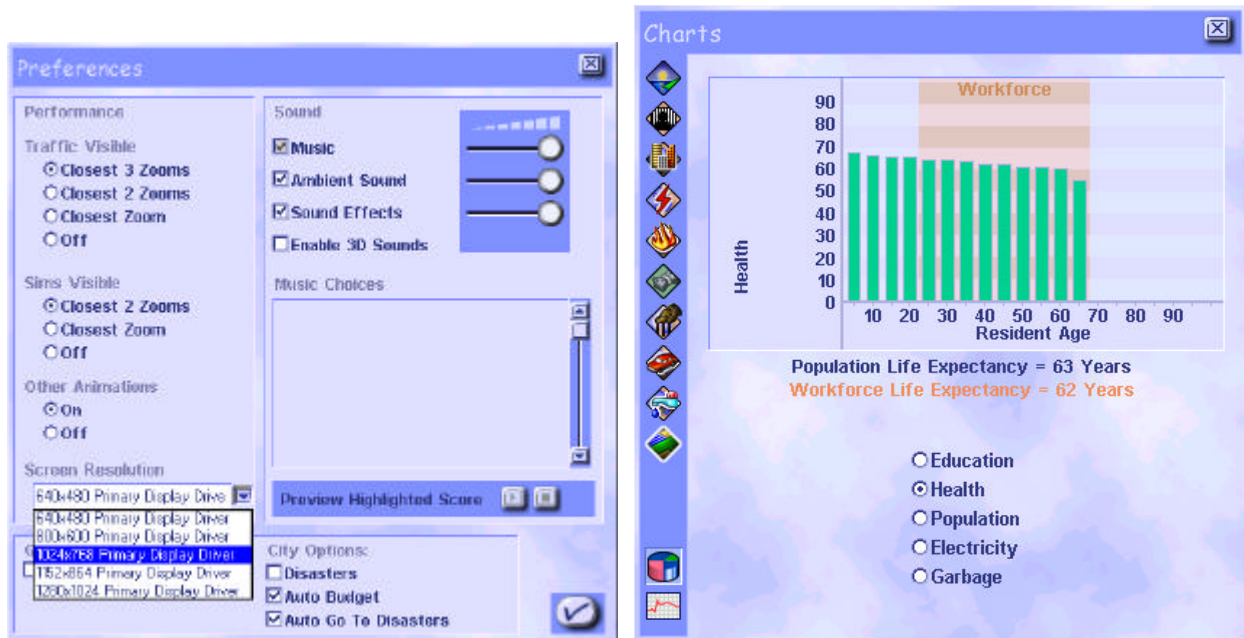
Remember that just because windows are mundane rectangular entities in Microsoft Windows doesn't mean they have to be that way in games. You can define a window to look and act practically any way you want. Here is a screenshot from SimCopter, a game we did a couple years ago. The user interface was done with a windowing system similar to the one described above. The windowing system was efficient enough that we decided to use it to implement the helicopter cockpit as well. When you write your own windowing system, you don't have to make it nearly as dull as a system like the one in Microsoft Windows.



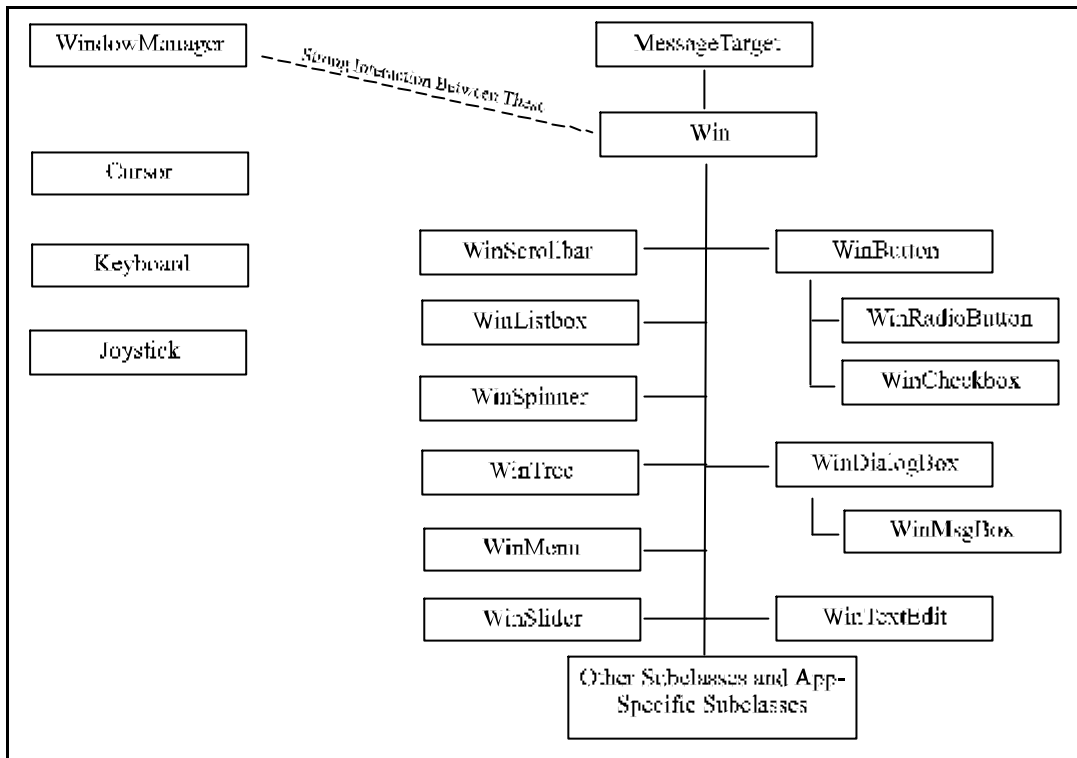
Now here are a few screenshots from our recently released SimCity 3000.



Here are a few more screenshots. I am not showing these to try to impress you with the windowing system we wrote, I am showing these to demonstrate what you can do in your games with a well-implemented windowing system. Virtually any game can benefit from this kind of system, regardless of the game type. The system can be very efficient and powerful.



Now let's examine the class hierarchy:



Here are the key points regarding this diagram and the system itself:

- The Win class is not the base class. The MessageTarget class is the base class. What's nice about this is that the messaging system is not limited to windows alone. Any class that inherits from MessageTarget can now receive messages. This is a powerful paradigm that Microsoft didn't implement in designing their windowing system. Hence, people started making all these invisible windows to receive messages from the system. In fact, Winsock itself is built on this hack. Here is the complete declaration:

```

struct IMessageTarget{
    virtual int DoMessage(int msg1, int msg2=0, int msg3=0, int msg4=0)=0;
};

```

That's it. Nothing more needed nor wanted.

- The Win class itself is where most of the code is. Here is a pared down version of the window class declaration:

```
class Win : public IMessageTarget{
public:
    virtual int          DoMessage(int msg1, int msg2=0, int msg3=0, int
msg4=0);
    virtual bool         SetPosition(int x, int y);
    virtual bool         SetArea(Rect2D& newArea);
    virtual int          Draw();
    virtual void         SetParent(Win* newParent);
    virtual Win*         GetParent();
    virtual bool         StartCapture();
    virtual bool         ReleaseCapture();
    virtual void         Hide();
    virtual void         Show();
    virtual const string& GetName();
    virtual void         SetName(const string&);
    virtual int          GetID();
    ... (about another 20 functions here)

protected:
    Rect2D area;
    int    id;
    string name;
    int    flags;
    Win*   parentWin;
    ... (a few extra data members here)
};
```

- The WindowManager class implements the housekeeping of the windowing system itself. It keeps track of what window has the focus and what window, if any, has the cursor captured. It implements keyboard and cursor message distribution, re-entrant modal dialogs, and interfacing with the operating system. The window manager has a (STL) hash-table of all valid windows, so if one window tries to delete another while the latter is busy, the window manager can postpone the actual deletion until the latter is ready. In this case, it puts the window in a "condemned" window list. There is a tight interaction between the Win and WindowManager class. The Win class declares the WindowManager class as a *friend*. All user input first comes through the WindowManager. The WindowManager decides, for example, what window is under the cursor, adjusts the cursor coordinates to be relative to the window, and then passes that window a cursor move message. Since the WindowManager keeps a list of all windows and knows the relationship between all windows, it also has functions such as:

```
FindWindow(int id)
ConvertPositionFromWinToWin(int x, int y, Win* winSrc, Win* winDest)
```

- The Cursor, Keyboard, and Joystick classes in the diagram above aren't formally part of the windowing system. In fact, they don't even know such a system exists. But the windowing system knows they exist and uses them. Most likely, the joystick would actually be used during gameplay directly by some entity in within the game loop. Here is the public interface of the Keyboard class:

```
class Keyboard{
public:
    bool  IsKeyDown(int key);
```

```
bool IsKeyDown(int key, int modifiers);
bool IsKeyDownNow(int key);
bool IsToggleKeySet(int toggleKey); //True if key (e.g. Numlock) is set.
int GetCurrentModifierState();      //Returns ORd value.
};
```

- Window controls are simply implemented by making subclasses of the `Win` class. This is no surprise, since many windowing systems work this way, such as OWL, MFC, TCL, and others. When a control receives a message it doesn't know how to deal with, it can pass the message up to its parent window. I said parent *window*, not parent *class*.
- This system may seem a little formal for something you think about using for a game, but let me tell you, this kind of system applies very well to just about any game.

Is a Circle a Valid Subclass of an Ellipse?

If you read *C++ FAQs*, by Cline and Lomow, you'll eventually come to a discussion that has caused much debate: *Is it proper to make a Circle a subclass of an Ellipse?* The authors' official answer: "Probably not." The reason why they argue this is based on one of the primary tenets of inheritance: Subclasses *extend* their parents. In geometry, it is convenient to think of a circle as a special case of an ellipse where the width and height are equal. A `Circle` would then be a class that is a restricted version of an `Ellipse`, and not an extension of it. For example, the `Ellipse` class probably has a `SetSize(x,y)` function. If the `Circle` is a subclass of `Ellipse`, then it can't really support this function. If anything, an `Ellipse` could be a subclass of a `Circle`. But that brings other problems.

My response to this is simply that this argument against making a `Circle` a subclass of `Ellipse` is basically sound and probably is a good rule of thumb to use. However, neither C++ nor any other language can perfectly describe or mimic the real world. So if it is convenient for you to make an `Ellipse` a subclass of a `Circle`, then do it. C++ is simply a tool.

public/private/protected

Just because it's offered and a lot of books out there talk about making classes with `private` or `protected` data and member access functions doesn't mean you have to do it that way all the time. Here is some code that someone (who will remain anonymous) wrote:

```
class Point3D{
public:
    float&      GetX()      { return x; }
    const float& GetX() const { return x; }
    float&      GetY()      { return y; }
    const float& GetY() const { return y; }
    float&      GetZ()      { return z; }
    const float& GetZ() const { return z; }
private:
    float x, y, z;
};
```

This is ridiculous. Making accessors for a basic structure like this provides no benefit. It's true that making the accessors inline will make their use as fast as direct access. But imagine having to use this class:

```
Point3D p1, p2;
float dist = ::sqrt(p1.GetX()*p2.GetX() + p1.GetY()*p2.GetY() + p1.GetZ()*p2.GetZ());
```

instead of:

```
float dist = ::sqrt(p1.x*p2.x + p1.y*p2.y + p1.z*p2.z);
```

All those function calls are annoying. The time to use protected data is when you really don't want the users of the class to either know about the data or modify the data directly. Here is an example of a good place to use protected data:

```
class Win{
public:
    virtual void SetArea(const Rect2D& rectNew) { rect=rectNew; InvalidateSelf(); }
    const Rect2D& GetArea(){ return rect; }

protected:
    Rect2D rect;
};
```

It's very useful to have accessors here, for a number of reasons. First of all, it would be bad for a user to simply poke values into the window's `rect` member. The window needs to know about position changes, so it can tell the windowing system that it has moved, so it can be redrawn. Secondly, you want to be able to make subclasses of the window, and they may want or need to be able to subclass the `SetArea` function.

Notice that we made the `Window::rect` member `protected` instead of `private`. You want to be very careful about making data `private` instead of `protected`. By making data `private`, you are basically making it so that users cannot make subclasses that can access that data. In many cases, this means users can't make subclasses at all. So by making data `private`, you are saying that nobody will ever want to make a subclass of your class. Why punish your users this way? You gain nothing. Personally, I *never* make anything `private`.

In conclusion, making data members `public` instead of `protected` is somewhat like using `goto`. Experienced and confident programmers know that there is a right time to use `goto`, and use it appropriately. Game programmers will likely feel comfortable using `public` data quite a bit. Consider this: all data members in C are public, and have been so for more than 20 years. Has this really caused much trouble?

Inlining

Inlining is critical to a good implementation of C++. It's so useful, in fact, that the C language standard has recently been modified to include the `inline` keyword as well. Inlining gives you all the efficiency advantages of preprocessor macros, yet are much easier to read and debug. Macros will continue to be useful for expression pasting. God knows that the Microsoft's authors of "windows.h" and MFC have probably written more lines of macros than actual regular code! I'll assume you already know the syntax of inlines.

The STL is generally implemented entirely inline. You can think of this as being good or bad. Too much inline code leads to cache thrashing, because the code is bulky, but too little inline code leads to cache thrashing, because all the function calls move the instruction pointer all over the code memory. Since most of the common functions in STL are small accessors and manipulators, you'll find that the inlining is beneficial. I can't tell you how important it is to keep the cache happy on Pentium processors when you're trying to write the most efficient code.

Virtually every current compiler I know of does a good job of inlining and can do recursive inlining to a decent depth. This includes Microsoft, Borland, Intel, Code Warrior, GNU, and Watcom compilers on multiple platforms. Use inlining liberally. That's all I have to say.

Pointers to Member Functions

Function pointers are often used in high-performance applications to give fast access to functions, particularly in graphics and language interpreters. Here we cover how to do this in C++. The declaration of pointers to C++ member functions has caused conniptions for programmers for years. Well, here's example code for how to do it for all kinds of member functions. Save this section, you will find it *very* useful in the future!:

```
struct A{
    A();
    ~A();
    void        Blah3();
    virtual void Blah4();
    int         Blah5(int x);
    virtual long Blah6(const char* c) const;
    static void Blah7(int x);
};

//Function pointer declarations
void (A::*pFunction1)()           = &A::A;           //Error. Impossible
void (A::*pFunction2)()           = &A::~~A;           //Error with VC++, but it
shouldn't be.
void (A::*pFunction3)()           = &A::Blah3; //
void (A::*pFunction4)()           = &A::Blah4; //Note that virtual functions
work fine
int (A::*pFunction5)(int)          = &A::Blah5; // and execute just as if they
were
long (A::*pFunction6)(const char*) const = &A::Blah6; // called by the normal
method.
void (*pFunction7)(int)            = &A::Blah7; //Notice that we don't use
"A::"
void (A::*pFunction3Array[10])(); //Declare array of function
pointers.

//Function pointer typedefs to match the above declarations with examples on the
right
typedef void (A::*Blah3PtrType)(); //Blah3PtrType pFunction3 = &A::Blah3;
typedef void (A::*Blah4PtrType)(); //Blah4PtrType pFunction4 = &A::Blah4;
typedef int (A::*Blah5PtrType)(int); //Blah5PtrType pFunction5 = &A::Blah5;
typedef long (A::*Blah6PtrType)(const char*); //Blah6PtrType pFunction6 = &A::Blah6;
typedef void (*Blah7PtrType)(int); //Blah7PtrType pFunction7 = &A::Blah7;

//More examples of function pointer usage.
A a, *pA; //Declare an object and a pointer to an object
(a.*pFunction3)(); //Same as a.Blah1();
(pA->*pFunction3)(); //Same as pA->Blah1();
pFunction3Array[0] = &A:: Blah3; //Note that Blah4 is virtual and Blah3 isn't,
yet can
pFunction3Array[1] = &A:: Blah4; // be assigned to the same function ptr type.
Cool.
(a.*pFunction3Array[0])(); //Call the 3rd member of the array. Same as
a.Blah1();
(pA->*pFunction3Array[1])(); //Call the 2nd member of the array. Same as pA-
>Blah1();
```

This is very useful for making interpreters and dynamic command execution through a table of function pointers. In a real world situation, you would have a class that has a bunch of functions that are declared with the same parameters. Subclasses could add new functions (new commands for the interpreter).

Operator overloading

As you probably know, operator overloading lets you redefine operators (+, *, !=, etc.). I refer you to any of the currently recommended C++ books out there for the syntax and limits of operator overloading.

Here is a list of all operators you can redefine:

+	-	++	--	*	/	%
^	&		~	!	<	>
=	+=	-=	*=	/=	%=	^=
&=	=	<<	>>	>>=	<<=	==
!=	<=	>=	&&		->*	->
,	()	[]	new	delete	new[]	delete[]

Let's cut to the chase: you should only use operator overloading where it actually makes sense.

Overloading the "+" operator for the string class makes a lot of sense. It is easy to see that a statement like

```
string1 += string2;
```

concatenates `string2` onto `string1`. Defining the "--" operator for the string class doesn't really make much sense, so don't do it. One of the best uses for operator overloading is for allowing mathematical operations on objects that aren't built-in numerical types.

Consider the following code:

```
struct A{
    A(){ printf("A::A()\n"); }
    int a;
};
A operator +(const A& a1, const A& a2){
    return A(a1.a + a2.a);
}
A a1, a2, a3;    //Three objects created here.
a1 = a2 + a3;    //One intermediate object created here!
```

Here's what actually prints:

```
A::A()
A::A()
A::A()
A::A()
```

You have to be careful about things like this. If the intermediate object is big (anything more than 32 bits), then creating the temporary object can be expensive. In the above example, the statement "`a1 = a2 + a3`" can actually compile to very efficient code (presuming you remove the `printf` constructor), since the `struct A` is actually nothing but an `int`. But imagine a class like this:

```
struct Matrix{
    float values[4][4];

    Matrix();
    Matrix(float values[4][4]);
};
Matrix operator +(const Matrix& m1, const Matrix& m2){
    Matrix temp;
    <do matrix addition here>
```

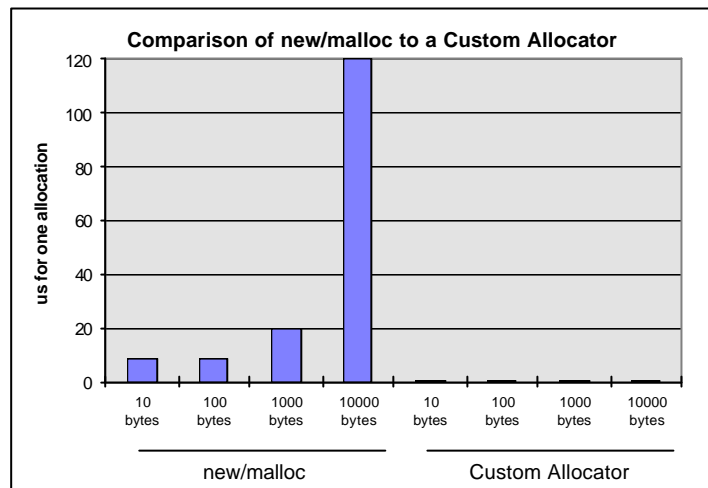
```
    return temp;  
}
```

Doing matrix operations this way is going to be too slow. You want to write custom routines to do matrix operations in the most efficient way. Just because you *can* do something in C++ doesn't mean that you *have* to. And no decent C++ programmer (games programmer or not) would think of doing 3D matrix math like the above example. You have to be concerned about two things when considering the efficiency of classes that are meant to be used with overloaded math operators:

- Temporaries. The operators that return temporaries will be a cause of slowdowns if the returned value is not something simple, like an `int`. The math operators that do this are: `+`, `-`, `*`, `/`, `%`, `|`, `&`, `^`, and `~`. Note that the operators that return by reference, `bool`, or `void` don't create these temporaries and are very efficient. They include `+=`, `-=`, `*=`, `/=`, `%=`, `|=`, `&=`, `^=`, `++`, `--`, `<`, `>`, `<<`, `>>`, and all the rest!
- Algorithms. Just like with the Matrix example above, you can sometimes implement a different and more efficient way of doing things than simply combining mathematical operations. Have you ever noticed that the Win32 SDK defines a function called `MulDiv(x, y, z)`? It returns the result of $x * y / z$. Why call this function when you can just write $x * y / z$. The answer is that most CPUs have special opcodes that allow you to do this operation more efficiently and without multiplication overflow.

Custom Allocators

One of the things about C++ that really benefits game programmers a lot is the ability to redefine `new` and `delete` both globally and on a class-by-class basis. This way, when you need to make allocation and deallocation go *really* fast, you can do it in a transparent way which allows the user to use `new` and `delete` on the class just like it was using the standard (slow) allocator. For those of you who haven't spent a lot of time examining how slow the standard `new/delete` and `malloc/free` are, here is an example of what kind of speed differences we're talking about here:



These kinds of allocators can be critical for high performance code. I have one of these in use in the SimCity 3000 graphics engine; it allocates and deallocates animating sprite records *very* fast, so animation can be as fast as possible. If you want cutting edge code, you *must* use custom allocators. Development libraries like SmartHeap use systems much like this.

I'm going to show you how to write a custom C++ allocator and give full example code for it. First of all, how do these allocators work? Here are the steps:

1. Allocate a memory block that can hold something like 32 equally sized objects.
2. These chunks are effectively made into a singly-linked list.
3. When an object is needed, the first item in the list is returned and the item is removed from the linked list.
4. When the object is freed, it is remade into a list item and put at the front of the list.

OK, now how do we write our own allocators in C++? First, here is how you do it globally (that is, replace the library implementation of `new`):

```
void* operator new      (size_t n) { return ::malloc(n); } //In our example, we
simply call
void* operator new[]   (size_t n) { return ::malloc(n); } //malloc and free.
void operator delete   (void* ptr) { ::free(ptr); }      //
void operator delete[] (void* ptr) { ::free(ptr); }      //
```

That's all there is to it. Now here is how to do it on a class-by-class basis:

```
struct A{
    ...
```

```

    void* operator new      (size_t n);           //For allocating a single object.
    void* operator new[]   (size_t n);           //For allocating an array.
    void operator delete   (void* ptr, size_t n); //Notice that in this version of
delete,
    void operator delete[](void* ptr, size_t n); //you must declare the size_t 2nd
argument.
    ...
};
void* A::operator new      (size_t n)           { <allocate it here> }
void* A::operator new[]   (size_t n)           { <allocate it here> }
void A::operator delete   (void* ptr, size_t n){ <free it here> }
void A::operator delete[](void* ptr, size_t n){ <free it here> }

```

Now what we need to do is provide the actual code to do the allocation and de-allocation. Here is the code:

```

class FastFixedAllocator{
public:
    FastFixedAllocator(unsigned int n);
    ~FastFixedAllocator();
    void* Alloc();
    void Free(void* pAlloc);

protected:
    struct Link{ Link* next; };
    struct Chunk {
        enum { size = 8*1024-16 };
        Chunk* next;
        char mem[size];
    };
    Chunk* chunks;
    const unsigned int esize;
    Link* head;
    void Grow();
};

inline void* FastFixedAllocator::Alloc(){
    if(head==0)
        Grow();
    Link* p = head;
    head    = p->next;
    return p;
}

inline void FastFixedAllocator::Free(void* pAlloc){
    Link* p = static_cast<Link*>(pAlloc);
    p->next = head;
    head    = p;
}

inline FastFixedAllocator::FastFixedAllocator(unsigned int sz)
: esize(sz<sizeof(Link*) ? sizeof(Link*) : sz)
{
    head    = 0;
    chunks = 0;
}

inline FastFixedAllocator::~FastFixedAllocator(){
    Chunk* n = chunks;
    while(n){
        Chunk* p = n;
        n = n->next;
        delete p;
    }
}

```

```

}
inline void FastFixedAllocator::Grow(){
    Chunk* n = new Chunk;
    n->next = chunks;
    chunks = n;

    const int nelem = Chunk::size/esize;
    char* start = n->mem;
    char* last = &start[(nelem-1)*esize];
    for(char* p = start; p<last; p+=esize)
        reinterpret_cast<Link*>(p)->next = reinterpret_cast<Link*>(p+esize);
    reinterpret_cast<Link*>(last)->next = 0;
    head = reinterpret_cast<Link*>(start);
}

```

Now, let's rewrite the new and delete operators for A to take advantage of this class:

Now what we need to do is provide the actual code to do the allocation and de-allocation. Here is the code:

```
FastFixedAllocator allocator(sizeof(A));  
void* A::operator new      (size_t n)          { return allocator.Alloc(); }  
void  A::operator delete  (void* ptr, size_t){ allocator.Free(ptr);      }
```

The implementation of the array ([]) new and delete is left as an exercise to the reader! (read, the author of this manual didn't quite have enough time to do this).

There is one gotcha that you need to know about when using this kind of "fixed size" allocator. If you make a subclass of A, and create members of it with new, A's version of new will get called. If the subclass is bigger than A, we have a problem. There are a number of workarounds:

- Simply don't subclass from A. This is the least desirable solution, but on the other hand, the number of times where you want a really fast allocator like this may not be often enough that you run into this problem a whole lot.
- Simply redefine new and delete for the subclass as well. You can even make them inline and call the global new and delete if you want.
- Use a more flexible version of the fixed allocator that isn't limited to a single size, but rather implements a number of fixed-block heaps and chooses the right one based on the size of the allocation. Choose increments that are a power of 2 so you can use & and | tricks to determine the right block very fast.

STL

STL ("Standard Template Library") is one of the coolest things in all of the C++ standard library, right up there with the string class. It is the standard C++ method for implementing containers. It consists of the following templated classes:

Container	Description
vector	Implements an array
list	Implements a doubly-linked list
queue	Implements a typical queue
deque	Implements a typical dequeue (double-ended queue)
priority_queue	Implements a queue with priorities assigned to each item
stack	Implements a standard stack
map	Implements a single-key associative container
multimap	Implements a multiple-key associative container
set	Implements a container of unique objects
multiset	Implements a container of non-unique objects
hash_set*	Implements a fast-access (but unsorted) set
hash_multiset*	Implements a fast-access (but unsorted) multi-set
hash_map*	Implements a fast-access (but unsorted) map
hash_multimap*	Implements a fast-access (but unsorted) multi-map
bitset	Implements a bit set, for compact data.
string	Implements a string class
valarray	Implements a vector specialized for numerics

* The hash containers were simply introduced too late to become part of the C++ standard, though their interface and behavior is well-defined enough that they can be considered a pseudo-standard. SGI provides a good implementation, but Microsoft does not. However, the hash containers are entirely built upon the other standard containers, so in theory, you can use the SGI hash classes with the Microsoft version of the other STL classes.

You can refer to the 1998 CGDC Conference Proceedings Document (on the distribution disk) for an analysis of the performance of the classes compared to hand-coded (e.g. list) or language-native (e.g. array) containers. If you don't feel like cracking that book open, let me tell you here: STL containers are *very* fast. They're fast enough to be used practically everywhere in the game, including the rendering subsystem. For example, accessing an STL `vector` is *exactly* as fast as accessing a traditional C array. Plus, the STL array has provisions for dynamic resizing when needed, and so on. At Maxis, we find the `vector`, `list`, `map`, and `hash_map` to be invaluable in our game programming.

What we're going to talk about here is how to make a custom allocator for STL. One of the neat things about STL is that you can associate an allocator with a container. So if you want a given STL container to allocate container objects from a faster memory allocator than `new`, you can do so. Note that due to the design of STL, you don't simply rewrite `operator new` for the class being contained. There are a couple logical reasons for this, actually. It allows classes being contained to not have to know they are being contained, it allows assignment of a single allocator to a number of classes, and it allows the container (and possibly the entire container library, as with SGI STL) to make a default custom allocation system tuned for the container (or entire container library).

Here's how you do it with the `FastFixedAllocator` we implemented above:

```
template <class T>
struct CustomSTLAllocator{
    T*      address(T& t)      const {return (&t); } //This is a slightly strange
    const T* address(const T& t) const {return (&t); } //required function. Just copy it.
    static T* allocate(size_type n){
        if(n==1)
```

```

        return static_cast<T*>(ffa.Alloc());
    }
    return static_cast<T*>(::malloc(n*sizeof(T)));
}
static void    construct(T* ptr, const T& value) { new(ptr) T(value); }
static void    deallocate(T* ptr, size_type n){
    if(n==1)
        ffa.Free(ptr);
        ::free(ptr);
    }
static void    destroy(T* ptr) { p->~T(); }
static size_type max_size() { return (size_type)-1; }

protected:
    static FastFixedAllocator ffa;
};
template<class T>
inline bool operator==(const CustomSTLAllocator<T>&, const CustomSTLAllocator<T>&) { return true; }
template<class T>
inline bool operator!=(const CustomSTLAllocator<T>&, const CustomSTLAllocator<T>&) { return false; }
FastFixedAllocator CustomSTLAllocator<int>::ffa(sizeof(int));

```

Notice that we make some of the functions above `static`. This is for backward compatibility with the old (mid-1997 and earlier) version of SGI STL. Since the standard library got hammered down in 1997, these functions normally aren't static. But you can make them static if you want, like we do above. The highlighted code above is the part of this class that is specific to the `FastFixedAllocator`. If you simply delete the highlighted code, you'll have a version which is fully generic. You can substitute any memory allocation routine you want here. So if you want to make your own custom STL allocator, just copy the above code into your project. Since it is templated, it can work with a any STL container of any C++ internal or user-defined class.

String Class

The string class has got to be one of the most anticipated features of the C++ language. Game programmers implementing user interfaces will definitely appreciate the string class. Say goodbye forever to memory allocation and overrun problems associated with C-style `char` array strings. Here are some of the cool things about the string class:

- Getting string length is very fast, due to structured array definition.
- Access to individual characters of the string is very fast—just as fast as with a C-style `char` array.
- A string class object can be treated as a NULL-terminated C-style `char` array by simply using `c_str()`.
- Concatenations can transparently resize the string as needed.
- There are a full set of substring and searching semantics like with other languages.
- Assignments can use copy-on-write techniques to make assignments very fast.

There are a couple deficiencies of the string class, due to academic design considerations of the language designers:

- `sprintf` is not directly possible with a C++ string.
- `toupper`, `tolower`, and other locale-specific functions are not directly possible with a C++ string. You want to be able to say something like this:

```
myString.toupper();
```

But this is not possible.

Here's how you get around these problems: you make a subclass of `string` that does whatever you want. You probably will want to make a subclass of `string` anyway, in order to deal with localization issues. See the **Localization** section for the reason why.

```
class String : public string{
    void toupper() { for(iterator i(begin()); i<end(); i++) ::toupper(*i); }
    void tolower() { for(iterator i(begin()); i<end(); i++) ::tolower(*i); }
    int  sprintf(const char* format, ...){
        char bigBuffer[16384];
        va_list argList;    //Make sure you #include <stdarg.h> for this.
        va_start(argList, lpszFormat);
        int nChars = _vsprintf(bigBuffer, kSizeOfMaxFixedString, lpszFormat,
argList);
        va_end(argList);
        assign(bigBuffer, nChars);
        return nChars;
    }
    bool isalnum(int index) { return ::isalnum(at(index)); }
    int  stricmp(string& strCompare) {
        return ::_stricmp(c_str(), strCompare.c_str());
    }
    int  strnicmp(string& strCompare, int nCount) {
        return ::_stricmp(c_str(), strCompare.c_str(), nCount);
    }
    ...
};
```

Here are a couple extra suggestions working with the `string` class or your subclass of it.

- Make your string resource loading *separate* from your string subclass. In other words, you probably don't want to make your string subclass have to know about your resource allocation system. This is simply because you want to be able to use this string class on all your company's projects, and I can guarantee you that your resource loading system will change in the future. We harp on this concept a number of times here: *always* make your low-level classes separate from your resource loading system. I suggest that you make a function called `StringFactory()` that takes your resource ID type and a string reference as an argument and makes a valid string out of it or returns false.
- Get familiar with the `string` class now. It's so easy. I've seen a number of new programmers who are used to C-style strings not understanding how simple it is to use C++ strings. They are so used to the C way of doing things that they try to apply that paradigm to C++ strings. For example, a new programmer a few months ago actually wrote this code (I'm not making this up):

```
String tempString1, tempString2;
char array[1024];
strcpy(array, tempString1.c_str());
strcat(array, tempString2.c_str());
tempString1 = array;
```

When all he had to do was this:

```
String tempString1, tempString2;
tempString1+=tempString2;
```

- SGI STL now provides a class called "rope", which is intended as a super string class (not as a replacement). It implements a string as a list of smaller substrings. This allows for making large text manipulation systems. For example, insertions are *much* faster with `rope` than with `string`.

Auto_ptr

The standard C++ library provides a templated class called `auto_ptr`. This is pretty nice because it allows you to make cleaner functions when those functions create a lot of pointers but need to bail out and clean up. Here's an example of such a function:

```
bool System::Init(){
    char* pData1=NULL, pData2=NULL; //Both need to be deleted on exit.
    bool bResult=false;
    pData1 = new char[DATA_1_SIZE];
    if(InitData1(pData1)){
        pData2 = new char[DATA_2_SIZE];
        if(InitData2(pData2)){
            if(UseData1AndData2(pData1, pData2)){
                bResult = true;
                goto OK;
            }
        }
    }
    TRACE("Error in System::Init()");
    DealWithError();
OK:
    delete pData1;
    delete pData2;
    return bResult;
}
```

You've seen this kind of thing before. You're either stuck with the "goto" in the middle of the function, or you make the `delete` calls in middle as well and `return true`. Sometimes the code can get pretty messy, due to all the logic and error checking. With `auto_ptr`, you can get around this problem in an efficient and clean way. Here's the re-written version of the function:

```
#include <memory> //auto_ptr is declared in <memory>.
bool System::Init(){
    auto_ptr<char>pData1(new char[DATA_1_SIZE]);
    auto_ptr<char>pData2(new char[DATA_2_SIZE]);
    if(InitData1(pData1) && InitData2(pData2)){ //Notice that we use pData1 and
        if(UseData1AndData2(pData1, pData2)) //pData2 exactly as if it were
    char*.
        return true;
    }
    TRACE("Error in System::Init()");
    DealWithError();
    return false;
}
```

What's nice about this system is that `pData1` and `pData2` will be deleted automatically on exit, even if an exception is encountered after their creation. The speed of the two functions above will be very similar to each other. `auto_ptr` is nice for these situations. I just find that a lot of time I have these `Init` functions that have a lot of the kind of logic that I show in the above example.

Exception Handling

True exception handling has been considered a sorely needed addition to the C language for "mission-critical" development. History has shown that other languages, such as Ada, are better for this kind of task. C's and C++'s power, and hence their popularity among game programmers, is that they let you write code that's pretty close to the machine itself. The language definition also allows very easy interfacing to assembly code as well. Exception handling, however, is a mechanism that doesn't map very simply to any low-level machine operation. Perhaps that's why it wasn't originally designed into C.

How does exception-handling work? Technically, the exception-handling mechanism is "implementation-specific." But, like most of C++'s "implementation-specific" features, 90%+ of the compilers implement exception handling in more-or-less the same way. The differences often come in if and how the exception handling mechanism interacts with the operating system. We have only enough space here to briefly examine the mechanism. For a good article on the details of how exception handling is implemented by C++ compilers under Win32, see Matt Pietrek's article in MSJ, Vol. 12, #1, Jan. 1997.

Basically, when you enter a function, a structure that contains the address of code that destroys stack-based objects is pushed onto the stack. The setup, push, pop, and shutdown of this structure takes up a number of machine instructions. However, with any decent compiler, this setup code only gets generated if the function creates automatic objects that have explicit destructors. Our experience has shown that most of the time-critical functions in a game app don't create these kinds of objects, and thus tend to not have this extra setup code.

Consider the following code:

```
struct A{
    A();
    ~A();
};
void TestFunction(){
    A a;
}
```

Here is the disassembly of the function with exception handling disabled:

```
7:      void TestFunction1(){
      push      ebp
      mov       ebp,esp
      sub       esp,00000004
8:      A a;
      lea       ecx,dword ptr [a] //Notice here that the compiler is putting
      call      A::A (00401290)    //the "this" pointer in the ecx register
9:      }                          //instead of pushing it onto the stack.
      lea       ecx,dword ptr [a]
      call      A::~A (004012a0)
      mov       esp,ebp
      pop       ebp
      ret
```

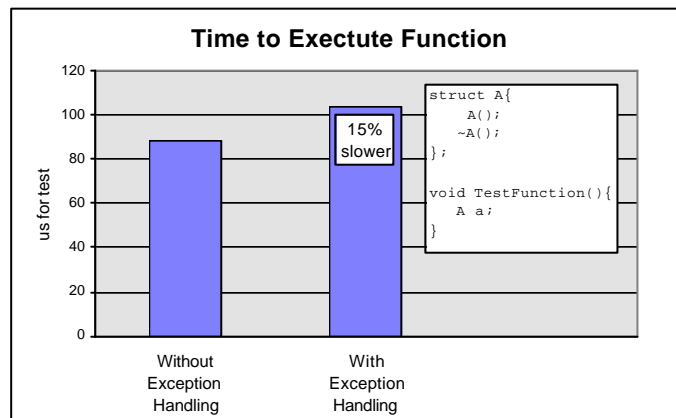
Here is the disassembly of the function with exception handling enabled. The highlighted code is the code added by the compiler to support exception handling:

```

7:    void TestFunction1(){
    push    ebp
    mov     ebp,esp
    push    ffffffff
    push    00401304
    mov     eax,dword ptr fs:[00000000] // Save the previous exception
    push    eax                        // registration structure.
    mov     dword ptr fs:[00000000],esp // Push the current exception
    sub     esp,00000004               // registration handler for
8:    A a;                             // this function.
    lea     ecx,dword ptr [a]
    call    A::A (004013b0)
9:    }
    mov     dword ptr [ebp-04],fffffff
    call    $L25934 (004012fc)         // Jump down to the destructor
    mov     eax,dword ptr [ebp-0c]
    mov     esp,ebp
    mov     dword ptr fs:[00000000],eax // Restore previous exception
    pop     ebp                       // registration handler.
    ret
$L25934:
    lea     ecx,dword ptr [ebp-10]
    jmp     A::~~A (004013c0)
$L25933:
    mov     eax,0040dc20
    jmp     ___CxxFrameHandler (004014d0)

```

Here are the results of timing the two versions:



For most game programmers, this is just too many instructions to spend preparing for something that almost never happens. Yet there are times when exception-handling is very useful, such as when you are doing a number of divisions and don't want to have to compare the denominator to zero before every one. You can simply let the exception handling mechanism catch the rare cases. So what is one to do? It turns out that some compilers allow a happy medium: exception handling enabled but automatic variable destruction stack unwinding disabled. With MSVC++, you simply disable exception handling's stack unwinding in the compiler options dialog box or on the command line with "-GX-" (the dash at the end means "disable"). This way, you can use `try...catch` statements, but when an exception happens, the `catch` works, but stack unwinding doesn't happen.

My recommendation for most situations is to use exception handling only sparingly. Use `try...catch` statements where you "expect" exceptions to happen, and not as a replacement for function return

values or as a replacement for `gotos` (you read it right), and definitely not to catch programming bugs. I've found that when programmers get carried away with exception handling, the code becomes hard to maintain, despite the fact that it may be "academically correct." What we like to do at Maxis is put one `try` statement around the entire game loop. If an exception happens, then give the user an option to save the game, and quit the app. Experience has shown that 90% of the time, the game save works fine. Users love this.

One last important thing to know about C++ exception handling: it doesn't always work! You heard it right. Consider the following code:

```
void FunctionThatTrashesExceptionHandling(){
    A    a;        //'A' is a class that has an explicit destructor.
    int x[4];
    for(int i=0; i<40; i++) //Trash the stack.
        x[i] = 0;
    *((char*)0) = 0; //Do something that causes an exception.
}
try{
    FunctionThatTrashesExceptionHandling();
}
catch(...){
    printf("This catch will never happen!\n");
}
```

The program will bomb and the exception will never get caught (at least it didn't on the Windows95 test machine). Because the exception handling mechanism uses the stack, the mechanism will fail if the stack gets corrupted. This lends credence to the belief that exception handling shouldn't be used to catch bugs.

Early versions of Microsoft Flight Simulator used to install an exception handler so that it didn't have to check for division overflows during runtime. If an overflow occurred, the given calculation was fudged or just skipped. This allowed it to get away with less range checking, thus being faster (and more reliable).

Run-Time Type Identification

RTTI (Run-Time Type Identification) allows you to query an object pointer or reference at runtime for its type. This is a very cool thing, at least in an academic sense of the word. Unfortunately, games are usually better off not using it for most situations. RTTI brings along both memory and speed penalties that can usually be avoided by designing your system differently. But just for entertainment value, and because it's so cool, I'll give some example code of what you can do with RTTI:

```
class A{
    virtual void DoSomething(); //Note that a class can't be part of RTTI unless it
};                                     //has at least one virtual function.
class B : public A{
    virtual void DoSomething();
};
class C : public A{
    virtual void DoSomething();
};
class D{                                     //Note that class D is not related to the above
    virtual void DoSomething();             classes.
};
```

Here's how you use the `typeid` operator to test the type of an object directly.

```
void TestIt(A* pA){
    if(typeid(pA) == typeid(A))
        printf("A\n");
    if(typeid(pA) == typeid(B)) //Note that you can use 'pA'
        printf("B\n");         // or
    if(typeid(*pA) == typeid(C)) //you can use '*pA'
        printf("C\n");
    if(typeid(*pA) == typeid(D))
        printf("D\n");
}
```

Here's how you use the `dynamic_cast` operator to cast an object:

```
void CastIt(A* pA, A& refA){
    B* pB = dynamic_cast<B*>(pA); //Null returned if not OK.
    if(pB)
        printf("Cast from pA to pB OK. pA points to a B.\n");
    else
        printf("Cast from pA to pB not OK. pA doesn't point to a B.\n");

    try{
        B& refB = dynamic_cast<B&>(refA); //Exception thrown if not OK.
        printf("Cast from refA to B OK. refA is not also a B.\n");
    }
    catch(...){
        printf("Cast from refA to B not OK. refA is also a B.\n");
    }
}
```

Here's how you use the `dynamic_cast` operator to cast an object:

```
void PrintIt(A* pA){
    printf("%s\n", typeid(*pA).name);
}
```

This is all you need to use RTTI. You'll find that with most compilers, you'll need to enable it as a compiler option if you want it. This is a good thing, because you can remove all RTTI memory overhead by simply disabling the option.

Stream IO

When you first start learning C++, your teacher or the book you're reading often starts off by showing you these two things called `cout` and `cin`. Many of the console application examples will look something like this:

```
void main(){
    cout << "hello world!" << endl;
}
```

I don't know about you, but I never liked this stuff. Yet for some reason, most C++ authors think C++ stream IO is the greatest thing and that `printf()` and its brethren should be banished from the standard library. Luckily, this banishment will likely never happen. And given the example code and benchmarks I'll show below, that's a good thing. Since we're game programmers, we have a history of doing things whatever way we want. We can certainly dump a library if it's not as fast as something else, no matter how elegant or inelegant the implementations. Remember, C++ is simply a tool. Use what works for you and ignore the rest.

The two basic uses of stream IO for most games programmers are to do string formatting and to do file IO. We'll examine both of these and present benchmarks comparing stream IO methods to the "classic" methods of doing the same thing. C++ strings are given additional coverage in a later section. You will see that while I (and I *do* speak for Maxis here) am not a fan of C++ stream IO, I am a fan of the C++ string class itself.

C++ Stream IO for String Formatting

You don't see a lot of examples of string formatting in most C++ texts, because it is a relatively recent addition to the C++ standard library. Nevertheless, it has been around for a few years now and a brief description of it can be found in the excellent third edition of *The C++ Programming Language*, by Bjarne Stroustrup, section 21.5.3. Perhaps another reason you don't see a lot about the string stream IO system is that it is so painful to use, navigate, and understand.

Here are some examples of `stringstream` use:

`printf`-based string output formatting

```
#include <stdio.h>
char charArray[128];
sprintf(charArray, "The player %s has %d points.\n", "Bill", 320);
```

`ostringstream` equivalent:

```
#include <iostream>
using namespace std;
ostringstream ost;
ost << "The player " << "Bill" << " has " << 320 << " points.\n";
//You can now access the string stored in ost by calling ost.str().
```

The `ostringstream` output example above is harder to read than the `sprintf` example, and it runs about 25%-50% slower. It's likely to be harder to maintain as well, since interpreting what's going on is harder due to all the extra symbols. Actually, the `ostringstream` example above is about as clean as they get. Try doing number formatting with any kind of C++ stream and you'll want to give up

programming altogether and take up something easier, like brain surgery. In preparing the example code for this section, I tried to find a way to clear out the above `ostream`. An hour later I gave up; perhaps it can't be done. That you can't do such a simple thing as this speaks for the ugly mess that C++ stream IO is. Now here's an example of `istream` input:

printf-based string input formatting:

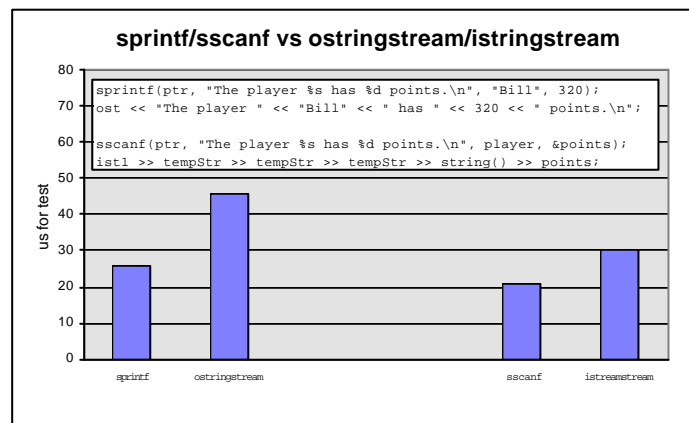
```
#include <stdio.h>
char charArray[128] = "The player Bill has 320 points.";
char player[32];
int points;
sscanf(charArray, "The player %s has %d points.\n", player, &points);
```

stringstream equivalent:

```
#include <iostream>
using namespace std;
stringstream ist(string("The player Bill has 320 points."));
string player, tempStr;
int points;
ist >> tempStr >> tempStr >> player >> tempStr >> points;
```

Since C++ `istream`s want to read from objects, and not arbitrary data, the only built-in way to emulate the simple `sscanf()` call is to do what we did above. The example speaks for itself. It runs about 50% slower than the `sscanf()`. It is clear that I am trashing the design of the C++ streams here. While this may sound to some like merely personal opinion, empirically, a large majority of game programmers despise C++ streams. The above benchmarks and example both explain and justify this point of view. Nevertheless, C++ streams do have one advantage: IO can be defined for user-defined types. While `printf()` only knows about built in primitive types such as `int`, `char*`, and `float`, C++ streams can be overloaded with virtually any type. Game programmers rarely have a use for this kind of feature. In a later section we will inspect another novel feature of the C++ standard library, the STL, and come to a different conclusion.

Here is a graph of the code and results described above. While this is just one example, tests have shown that the results below are representative of the results you'll get with other cases.



C++ stream IO for file operations

Here's an example of how to open a binary file and read in some data via C++ stream IO:

```
#include <fstream.h>
ifstream stream;
char data[1024];
int read_count;
stream.open("C:\\blah.dat", ios::in | ios::binary);
```

```

if(stream.is_open()){           //Note that open() returns void.
    stream.read(data, 1024);    //Note that read() doesn't return count
    read_count = stream.gcount(); //but instead returns stream&.
    stream.close();
}

```

You can also do C++ file IO with the streaming operators << and >>. Here is an example of how you would do this:

```

#include <fstream.h>

class A{
    long a;
    friend istream& operator>>(istream& is, A& a);
};

istream& operator>>(istream& is, A& a){
    is >> a.a;
    return is;
}

ifstream stream;
A          a;
int        read_count;
stream.open("C:\\\\blah.dat", ios::in | ios::binary);
if(stream.is_open()){           //Note that open() returns void.
    stream >> a;
    read_count = stream.gcount();
    stream.close();
}

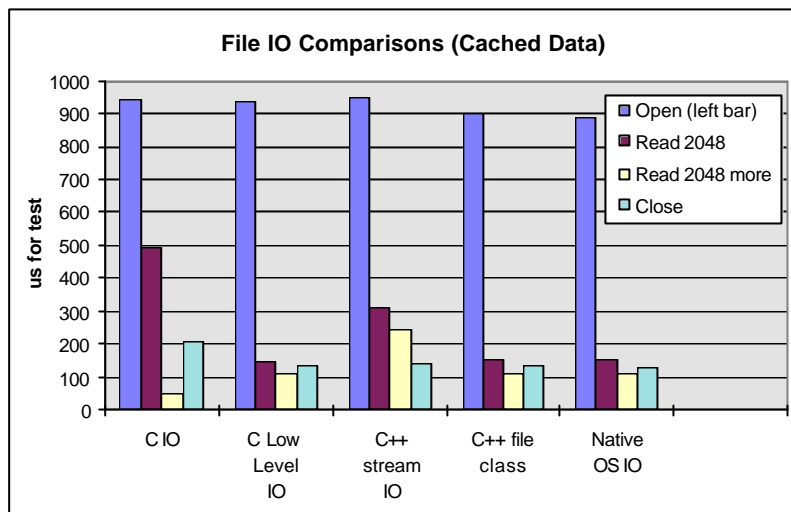
```

C++ file IO via the streaming operators << and >> is not very friendly to the concept of reading in arbitrary binary data. The streaming operators need the operands to be classes, and those classes have to specifically have << and >> defined for them and the IO class. Yes, you can get around this, but to do so is messy and tedious. It's usually just easier to use `fstream::read()` and `fstream::write()` to do your work.

Let's get to the important part. How fast is the C++ stream IO compared to other methods? Here we do a simple comparison of five methods for opening a file, reading it, and closing it. When doing such comparisons, care needs to be taken that you account for file buffering that the operating system or run time library may be doing for you. Here are the five methods:

Method	Description/Example
C IO	-- <code>fopen()</code> , <code>fread()</code> , ...
C low level IO	-- <code>_open()</code> , <code>_read()</code> , ...
C++ stream IO	-- <code>ifstream</code>
C++ home-grown file class IO	-- C++ wrapper around native OS IO
Native OS IO	-- <code>Win32 CreateFile()</code>

Here are some comparisons of cached file operations between the above-mentioned methods, after averaging many runs:



File opening took roughly the same time for all methods. File closing consistently took a little longer for C IO. All reading was slow for C++ stream IO, whereas the caching system of C IO caused first reads to be slow, but subsequent reads to be fast. C low level IO, C++ file class, and native OS all reads took basically the same amount of time. It is important to note that the results presented above represent the results of the using the C and C++ standard libraries of MSVC++ on Windows 95 and Windows NT. Other OSs and libraries may yield different results.

To my surprise, the C++ stream IO was consistently significantly slower than all other methods. Examining the source and disassembly of the various methods showed that the `ifstream` IO, under Visual C++, simply spent a lot of time doing housekeeping. The C IO and C low level IO methods basically spent a little time calling thread-safety functions, then called the native OS method to do the actual work. This explains why they were close in speed to the Native OS IO and home-grown C++ class IO. I recommend using a home-grown file interface for game programming, as it provides the most flexibility and speed, yet provides a platform-independent interface.

The take-home message here is that, unless the C++ stream IO classes are providing you with some extra necessary benefits, you probably want to stay away from them when writing high-performance code. Nowhere in the C++ language definition does it say you *must* use C++ stream IO. It is simply there if you want to use it. Like any other C++ feature, you can simply ignore it if you don't want to use it.

If you really want the highest-performance file IO, you might want to take advantage of two things: asynchronous IO and unbuffered IO. Asynchronous IO lets you start a file read in the background and let a callback function notify you when the read is complete. Unbuffered IO gives you fine control over exactly what bytes get read from the disk, making significant gains for random access file reads. I don't have space to give examples here, but the methods are not too difficult under Win32 or MacOS. Other OSs may vary.

Templates

The reason templates were invented was to facilitate the generation of generic container classes. That's about 75% of their usefulness, actually. In the next section, we see another use for them.

Two minute introduction to how to use templates.

Making templated classes is as easy as cake (eating it, not making it). Below we show a class in regular and templated forms side-by-side.

Non-Templated Version	Templated Version
<pre>struct Point3D{ int x; int y; int z; Point3D(){} Point3D(int nx, int ny, int nz) : x(nx), y(ny), z(nz){} bool IsInBox(const Point3D& pt1, const Point3D& pt2) { return x>=pt1.x && x<pt2.x && y>=pt1.y && y<pt2.y && z>=pt1.z && z<pt2.z; } };</pre>	<pre>template <class T> struct Point3D{ T x; T y; T z; Point3D(){} Point3D(T nx, T ny, T nz) : x(nx), y(ny), z(nz){} bool IsInBox(const Point3D& pt1, const Point3D& pt2) { return x>=pt1.x && x<pt2.x && y>=pt1.y && y<pt2.y && z>=pt1.z && z<pt2.z; } };</pre>

All you do to make the templated version is put "template <class T>" in before the class declaration, and replace all instances of "int" with "T". The highlighted text above is the only text that has changed from the left side. Pretend that every instance of "T" is simply replaced with "int" at compile time. You don't have to use "T;" you can use any letter or multi-letter identifier. "T" is just a common convention.

You can also use multiple types, like this:

```
template <class T, class A, int I, float F>
class Blah{
    T t;
    A a;
    enum { e = I };
    float Multiply() { return t*a*F; }
};
```

That's most of what you need to know about templates. The only other important things you might want to know is how to subclass from templates (trivial) or how to put template definitions (as opposed to declarations) in CPP files. See Stroustrup's *The C++ Programming Language*, 3rd Edition, chapter 13, for how to do these and more.

Example of Cool C++ Class

We're going to put our template knowledge and our operator overloading knowledge to create a very useful C++ class for fixed point math. Many of you are familiar with fixed point math done C or assembly style. It involves defining a fixed point type as a left-shifted `int` and making various macros to convert between various other numeric types. Fixed point math is very useful because:

- It allows mathematical operations that is as fast or faster than floating point, even on Pentium and Pentium Pro/II processors. This depends a lot on the operations and their order. Floating point multiplies can often be faster on Pentium and later machines.
- Conversions between `fixed` and `int` are much faster than conversions between `float` and `int`. Because of this, if you are writing code that must do floating point conversions to `int` frequently, you'll pay a penalty. Read the next bullet point for more on this.
- Trigonometry is much faster for fixed point than it is for floating point, on all processors. This is because you can make a very fast lookup table for fixed point, since the fixed point value can be converted to the lookup index with just a simple right shift. Floating point values must be converted to integers before doing a lookup, and that is *slow*. You can speed this up by providing a trick inline function to convert from `float` to `int` which is *much* faster than the standard `float` to `int` conversion. Here is the function:

```
inline int FloatToInt(float fValue){ //Takes 6 clock ticks on an Intel PII.
    double dTemp = fValue + (((65536.0 * 65536.0 * 16.0) + 32768.0) * 65536.0);
    return (*(int*)&dTemp) - 0x80000000;
} //Note, values will be rounded with this method, not chopped.
```

Nevertheless, trigonometry will still usually be faster by doing fixed point lookup tables. Cache pot-shotting (a.k.a. cache thrash) is the only potential problem fixed point trigonometry lookups.

Here is an example of C-based fixed point support code:

```
typedef int SFixed16; //Signed fixed point 16:16 (16 bits integer, 16 bits
fraction).
#define Fixed16ToInt(x)      (((int)(x) >> 16)
#define IntToFixed16(x)      (((Fixed16)((x) << 16))
#define Fixed16ToDouble(x)   (((double)x) / 65536.0)
#define DoubleToFixed16(x)   (((Fixed16)((x) * 65536.0))
#define Fixed16ToFloat(x)    (((float)x) / 65536.0F)
#define FloatToFixed16(x)    (((Fixed16)((x) * 65536.0F))
SFixed16 Fixed16Mul(SFixed16 x, SFixed16 y); //Implemented in ASM.
SFixed16 Fixed16Div(SFixed16 x, SFixed16 y);
```

If you've ever written fixed point math code in C that uses functions like those above, you know how it can be rather messy looking and sometimes confusing to decipher, given all the conversion calls you end up making. Wouldn't it be nice to have your fixed point data type act just like a built-in type, such as `float`?

OK, now it's time to define our fixed point class. This class is tough because we are trying to define a C++ type that acts like an internal numerical type and smoothly integrates with standard numerical types, such as `float`, `int`, and `char`. Reasons:

- C/C++ implement a distinct type promotion system when evaluating mathematical statements involving different numerical types. Here's a fun quiz question: what is the value of `x` after the following assignment?

```
float x = int(-1)*unsigned(1); //x = -1*1;
```

The answer is 4.29497×10^9 . The reason is that when you combine an `int` and an `unsigned int` in an equation, the `int` gets converted to `unsigned int`. And `-1` casted to `unsigned` is 4 billion-something. Now that you know this, you should be able to tell what the result is of this:

```
bool result = int(-1) < unsigned(1);
```

- Intermediate values are dealt with silently (and efficiently) by the compiler.
- There are at least 10 different mathematical types available (`char`, `uchar`, `short`, `ushort`, etc.), and we must make our class work with all of them transparently.

As a result, the fixed point class is going to have a lot of operators defined for it. Here's how you define a 16:16 version of the class:

```
typedef FPTemplate<int, (int)16, (int)16> SFixed16;
typedef FPTemplate<unsigned, (int)12, (int)20> UFixed12;
```

Note that we must supply the shifting values as integers passed into the template. We must also supply a custom version of the multiply and divide functions for each templated version. This is easy, because all we have to do is copy, paste, and modify a couple values from the 16:16 multiply and divide functions. Since the class merely wraps a 32 bit `int` type, the generated code is *very* efficient. Even functions that create temporaries aren't an efficiency problem because the temporary is nothing but an `int`, and compiler math statement code generation is well optimized to deal with integer (and float) intermediates, particularly since these types fit within a register. The full source code is available.

Full mathematical syntax is legal with this fixed point class. In other words, code like this works fine:

```
SFixed16 f;
double d;
int i;

float value = i*3+d/(f+1);
printf("%f", f.AsFloat());
```

You may ask, "Why do you make a function called `AsFloat()` instead of simply writing a float conversion operator for the fixed point class?" The answer is that if we write `operator float()`, then the compiler would get confused when writing statements with mixed types, like the `"i*3+d/(f+1)"` statement above. It would get confused because when it tries to compile a substatement like `"f+1"`, the compiler wouldn't know whether it should convert the `f` to an `int` or convert the `1` to fixed point. For statements that involve only built-in numerical types, the compiler has a set of promotion rules that are defined in the language standard. We simply don't have that luxury when defining our own type. This is the single biggest deficiency in operator overloading in C++.

Here is the header file for the class. This is a fairly rigorous implementation:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// class FTemplate
//
template <class T, int upShiftInt, int downShiftInt, int upMulInt, int downDivInt>
struct FTemplate{
    //Data
    T value;

    //Types
    typedef T type;

    //Functions
    FTemplate() {}
    FTemplate(const FTemplate& newValue) { value = newValue.value; }
    FTemplate(const int& newValue) { value = newValue << upShiftInt; }
    FTemplate(const unsigned int& newValue) { value = newValue << upShiftInt; }
    FTemplate(const long& newValue) { value = newValue << upShiftInt; }
    FTemplate(const unsigned long& newValue) { value = newValue << upShiftInt; }
    FTemplate(const float& newValue) { value = (int)(newValue * (float)upMulInt); }
    FTemplate(const double& newValue) { value = (int)(newValue * (double)upMulInt); }
    void FromFixed(const int& newValue) { value = newValue; }
    T AsFixed () { return value; }

    //We don't define the conversion operators because that would cause a lot of compiler
    // errors complaining about not knowing what function to call. Thus, we have functions
    // below such as "AsInt()" to do explicit conversions.
    //operator int() const { return (int) (value>>downShiftInt); }
    //operator unsigned int() const { return (unsigned int) (value>>downShiftInt); }
    //operator long() const { return (long) (value>>downShiftInt); }
    //operator unsigned long() const { return (unsigned long) (value>>downShiftInt); }
    //operator float() const { return (float) (value / (float)downDivInt); }
    //operator double() const { return (double) (value / (double)downDivInt); }

    int AsInt() const { return (int) (value>>downShiftInt); }
    unsigned int AsUnsignedInt() const { return (unsigned int) (value>>downShiftInt); }
    long AsLong() const { return (long) (value>>downShiftInt); }
    unsigned long AsUnsignedLong() const { return (unsigned long) (value>>downShiftInt); }
    float AsFloat() const { return (float) (value / (float)downDivInt); }
    double AsDouble() const { return (double) (value / (double)downDivInt); }

    FTemplate& operator=(const FTemplate& newValue) { value = newValue.value; return *this; }
    FTemplate& operator=(const int& newValue) { value = newValue << upShiftInt; return *this; }
    FTemplate& operator=(const unsigned int& newValue) { value = newValue << upShiftInt; return *this; }
    FTemplate& operator=(const long& newValue) { value = newValue << upShiftInt; return *this; }
    FTemplate& operator=(const unsigned long& newValue) { value = newValue << upShiftInt; return *this; }
    FTemplate& operator=(const float& newValue) { value = (int)(newValue * (float)upMulInt); return *this; }
    FTemplate& operator=(const double& newValue) { value = (int)(newValue * (double)upMulInt); return *this; }

    bool operator< (const FTemplate& compareValue) const { return value < compareValue.value; }
    bool operator> (const FTemplate& compareValue) const { return value > compareValue.value; }
    bool operator== (const FTemplate& compareValue) const { return value == compareValue.value; }
    bool operator!= (const FTemplate& compareValue) const { return value != compareValue.value; }

    bool operator< (const int& compareValue) const { return value < (T)(compareValue<<upShiftInt); }
    bool operator> (const int& compareValue) const { return value > (T)(compareValue<<upShiftInt); }
    bool operator== (const int& compareValue) const { return value == (T)(compareValue<<upShiftInt); }
    bool operator!= (const int& compareValue) const { return value != (T)(compareValue<<upShiftInt); }

    bool operator< (const unsigned int& compareValue) const { return value < (T)(compareValue<<upShiftInt); }
    bool operator> (const unsigned int& compareValue) const { return value > (T)(compareValue<<upShiftInt); }
    bool operator== (const unsigned int& compareValue) const { return value == (T)(compareValue<<upShiftInt); }
    bool operator!= (const unsigned int& compareValue) const { return value != (T)(compareValue<<upShiftInt); }

    bool operator< (const long& compareValue) const { return value < (T)(compareValue<<upShiftInt); }
    bool operator> (const long& compareValue) const { return value > (T)(compareValue<<upShiftInt); }
    bool operator== (const long& compareValue) const { return value == (T)(compareValue<<upShiftInt); }
    bool operator!= (const long& compareValue) const { return value != (T)(compareValue<<upShiftInt); }

    bool operator< (const unsigned long& compareValue) const { return value < (T)(compareValue<<upShiftInt); }
    bool operator> (const unsigned long& compareValue) const { return value > (T)(compareValue<<upShiftInt); }
    bool operator== (const unsigned long& compareValue) const { return value == (T)(compareValue<<upShiftInt); }
    bool operator!= (const unsigned long& compareValue) const { return value != (T)(compareValue<<upShiftInt); }

    bool operator< (const float& compareValue) const { return value < compareValue*(float)upMulInt; }
    bool operator> (const float& compareValue) const { return value > compareValue*(float)upMulInt; }
    bool operator== (const float& compareValue) const { return value == compareValue*(float)upMulInt; }
    bool operator!= (const float& compareValue) const { return value != compareValue*(float)upMulInt; }

    bool operator< (const double& compareValue) const { return value < compareValue*(double)upMulInt; }
    bool operator> (const double& compareValue) const { return value > compareValue*(double)upMulInt; }
    bool operator== (const double& compareValue) const { return value == compareValue*(double)upMulInt; }
    bool operator!= (const double& compareValue) const { return value != compareValue*(double)upMulInt; }
    bool operator! () const { return value == 0; }

    FTemplate operator~ () const { FTemplate temp; temp.value = ~value; return temp; }
    FTemplate operator- () const { FTemplate temp; temp.value = -value; return temp; }
    FTemplate operator+ () const { return *this; }

    FTemplate& operator+=(const FTemplate& argValue) { value += argValue.value; return *this; }
    FTemplate& operator+=(const int& argValue) { value += (T)(argValue<<upShiftInt); return *this; }
}

```

```

FPTemplate& operator+=(const unsigned int& argValue) { value += (T)(argValue<<upShiftInt); return *this; }
FPTemplate& operator+=(const long & argValue) { value += (T)(argValue<<upShiftInt); return *this; }
FPTemplate& operator+=(const unsigned long& argValue) { value += (T)(argValue<<upShiftInt); return *this; }
FPTemplate& operator+=(const float& argValue) { value += int(argValue*(float)upMulInt); return *this; }
FPTemplate& operator+=(const double& argValue) { value += int(argValue*(double)upMulInt); return *this; }
FPTemplate& operator+=(const FPTemplate& argValue) { value += argValue.value; return *this; }
FPTemplate& operator-=(const int& argValue) { value -= (T)(argValue<<upShiftInt); return *this; }
FPTemplate& operator-=(const unsigned int& argValue) { value -= (T)(argValue<<upShiftInt); return *this; }
FPTemplate& operator-=(const long& argValue) { value -= (T)(argValue<<upShiftInt); return *this; }
FPTemplate& operator-=(const unsigned long& argValue) { value -= (T)(argValue<<upShiftInt); return *this; }
FPTemplate& operator-=(const float& argValue) { value -= int(argValue*(float)upMulInt); return *this; }
FPTemplate& operator-=(const double& argValue) { value -= int(argValue*(double)upMulInt); return *this; }
FPTemplate& operator*=(const FPTemplate& argValue) { value = FixedMul(value, argValue.value); return *this; }
}
FPTemplate& operator*=(const int& argValue) { value = FixedMul(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator*=(const unsigned int& argValue) { value = FixedMul(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator*=(const long& argValue) { value = FixedMul(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator*=(const unsigned long& argValue) { value = FixedMul(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator*=(const float& argValue) { value = FixedMul(value, (type)(argValue*(float)upMulInt));
return *this; }
FPTemplate& operator*=(const double& argValue) { value = FixedMul(value, (type)(argValue*(double)upMulInt));
return *this; }
FPTemplate& operator/=(const FPTemplate& argValue) { value = FixedDiv(value, argValue.value); return *this; }
FPTemplate& operator/=(const int& argValue) { value = FixedDiv(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator/=(const unsigned int& argValue) { value = FixedDiv(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator/=(const long& argValue) { value = FixedDiv(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator/=(const unsigned long& argValue) { value = FixedDiv(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator/=(const float& argValue) { value = FixedDiv(value, (type)(argValue*(float)upMulInt));
return *this; }
FPTemplate& operator/=(const double& argValue) { value = FixedDiv(value, (type)(argValue*(double)upMulInt));
return *this; }
FPTemplate& operator%=(const FPTemplate& argValue) { value = FixedMod(value, argValue.value); return *this; }
FPTemplate& operator%=(const int& argValue) { value = FixedMod(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator%=(const unsigned int& argValue) { value = FixedMod(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator%=(const long& argValue) { value = FixedMod(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator%=(const unsigned long& argValue) { value = FixedMod(value, argValue<<upShiftInt); return *this; }
}
FPTemplate& operator%=(const float& argValue) { value = FixedMod(value, (type)(argValue*(float)upMulInt));
return *this; }
FPTemplate& operator%=(const double& argValue) { value = FixedMod(value, (type)(argValue*(double)upMulInt));
return *this; }

FPTemplate& operator|=(const FPTemplate& argValue) { value |= argValue.value; return *this; }
FPTemplate& operator|=(const int& argValue) { value |= (argValue<<upShiftInt); return *this; }
FPTemplate& operator&=(const FPTemplate& argValue) { value &= argValue.value; return *this; }
FPTemplate& operator&=(const int& argValue) { value &= (argValue<<upShiftInt); return *this; }
FPTemplate& operator^=(const FPTemplate& argValue) { value ^= argValue.value; return *this; }
FPTemplate& operator^=(const int& argValue) { value ^= (argValue<<upShiftInt); return *this; }

FPTemplate operator<<=(int numBits) const { return value << numBits; }
FPTemplate operator>>=(int numBits) const { return value >> numBits; }

FPTemplate& operator<=<=(int numBits) { value <= numBits; return *this; }
FPTemplate& operator>=>=(int numBits) { value >= numBits; return *this; }

FPTemplate& operator++() { value += 1<<upShiftInt; return *this; }
FPTemplate& operator--() { value -= 1<<upShiftInt; return *this; }
FPTemplate operator++(int) { FPTemplate temp(*this); value += 1<<upShiftInt; return temp; }
FPTemplate operator--(int) { FPTemplate temp(*this); value -= 1<<upShiftInt; return temp; }

FPTemplate Abs() { if(value<0) return -value; return value; }
FPTemplate DivSafe(const FPTemplate& denominator) { FPTemplate temp; temp.FromFixed(
FixedDivSafe(value, denominator.value)); return temp;
}
FPTemplate& DivSafeAssign(const FPTemplate& denominator) { value = FixedDivSafe(value, denominator.value); return
*this; }

static T FixedMul (const T t1, const T t2); // If you are using one of these functions directly or
static T FixedDiv (const T t1, const T t2); // indirectly through one of the above operators, you
static T FixedDivSafe (const T t1, const T t2); // need to define the function in a separate .cpp
file.
static T FixedMulDiv (const T t1, const T t2, const T t3); // Implementation may be processor- or compiler-
specific.
static T FixedMulDivSafe (const T t1, const T t2, const T t3); //
static T FixedMod (const T t1, const T t2); // 'Safe' versions of functions return maximum
possible
static T FixedModSafe (const T t1, const T t2); // value when overflow or division by zero would
occur.
};
////////////////////////////////////

```

High Performance C++

We could talk extensively here about performance and efficiency issues with C++. But I already have written 30 fairly dense pages of discussion about this in the 1998 GDC Conference Proceedings. This document is on the distribution CD for this class in the Documents directory as "*High Performance Game Programming in C++.doc*". I refer you to this document for an extensive and detailed discussion about the inner workings of C++ code generation. The basic take-home message of that document is this: C++ compilers and code generation is surprisingly efficient, so if you know what you're doing, you will get the best of both worlds: efficiency and power.

Mixing C, C++, and Assembly

There's a good chance that even if you are writing your games in C++, you'll have to interface with C and assembly code. You already be familiar with the weird linking errors you can get if you don't interface C++ to C correctly. This is largely do to *C++ name mangling*.

C++ name mangling

C++ was originally designed to be compilable under C. The first C++ "compilers" were C code generators. Because of this, C++ name mangling was invented. C compilers will generally do something like prepend an underscore to a function name before the function is inserted into the library symbol table. But since C++ allows function name overloading, a simple underscore won't do. C++ compilers generate "mangled" or "decorated" (more euphemistic) symbol table names to get around this problem.

Calling C from C++

To call C from C++, you use "extern "C"". This applies to both functions and data. Let's say you have a C source file (blah.c) and C header file (blah.h), and you want to reference variables and functions declared and defined in those files from C++. Here is what you do:

blah.h	blah.c	blah1.cpp
extern int global_value; void DoSomething(void);	int global_value; void DoSomething(){ global_value++; }	extern "C" int global_value; extern "C" void DoSomething(); global_value++; DoSomething();

or

blah.h	blah.c	blah1.cpp
#ifdef __cplusplus extern "C"{ #endif extern int global_value; } void DoSomething(void); #ifdef __cplusplus } #endif	int global_value; void DoSomething(){ global_value++; }	#include <blah.h> global_value++; DoSomething();

If the function you are calling a Pascal, FORTRAN, or Pascal-convention ASM function, you must also use "`__pascal`" or "`__stdcall`" in addition to "C". The syntax is compiler-dependent; see your documentation for details.

Calling C++ from C

To make C++ functions and variables visible from C, you must basically tell your C++ compiler to make the given variables and functions C variables and functions. Again, you use 'extern "C"'. C++ structs are also available from C, as long as they don't have member functions, inheritance, or any other C++-specific struct feature. Strange enough, referencing C++ structs from C generally doesn't require using 'extern "C"' on the struct--at least not on PC compilers. Here's an example:

blah.h	blah.cpp	blah1.c
extern "C" int global_value;	int global_value;	global_value++;
extern "C" void DoSomething();	DoSomething();	DoSomething(){
	global_value++;	
	}	
struct Blah{	Blah blah;	struct Blah blah;
int x;	blah.x++;	blah.x++;
};		

Note that there are some limitations with sharing C++ functions with C:

- You cannot declare a member function with extern "C".
- You can specify extern "C" for only one instance of an overloaded function; all other instances of an overloaded function have C++ linkage and subsequent name mangling.

Calling Assembly from C++

Calling assembly functions and using assembly global variables is just like calling C code.

Calling C++ from Assembly

Calling C++ functions and using C++ global variables from assembly is just like from C.

Inline Assembly with C++

Inline assembly in C++, much like inline assembly in C, is somewhat compiler- and processor-specific. Nevertheless, PC compiler vendors, and other compiler vendors as well have implemented a reasonably consistent extension using "asm" or "__asm". However, what each compiler accepts within the "asm" statement is often very specific to the compiler. Here are two examples of code that is valid with Microsoft, Symantec, Borland, and Intel C++ compilers. With CodeWarrior, only the second inline method is legal.

```
void DoSomething(){
    __asm mov eax, 4      ; ASM-style comments are legal
    __asm{                ; Code Warrior only accepts this
        mov ecx, 3        ; style of inline asm.
        inc ecx
    }
}
```

Inline assemblers also generally let you use both local variables and member variables, too. Here's an example of this for various compilers:

```
int DoAsm(int x){
    __asm mov eax, x
```

```

    __asm ret
}

```

Note that with the Microsoft, Intel, Borland, CodeWarrior, and other compilers, you can remove the function prolog and epilog code by prefacing the function with `__declspec(naked)`. This is particularly useful for functions that won't be calling any other functions. With Code Warrior, you must supply the "{asm ret}" yourself, because the compiler won't generate it for you (and the function will thus crash). Thus, the function above would be written like this:

```

__declspec(naked) int DoAsm(int x){
    __asm mov eax, x
    __asm ret
}

```

The documentation for some of the C++ compilers says that you can't call C++ class member functions from within inline assembly code. This is not true. What they mean to say is that you cannot *directly* call C++ member functions from inline assembly. If you simply take the address of the member function and assign it to a pointer variable, you can call it like any other function, just as long as you make sure to pass the "this" pointer appropriately to called member function. Here's an example with Microsoft VC++:

```

class A{
    int a;
    void DoA();
    virtual void DoAVirtual();
};

void TestAsmA(A* pA){
    int x;

    //x++
    __asm inc dword ptr[x]           //Increment a local variable

    //pA->a++;
    __asm mov eax, pA               //Increment pA->a
    __asm inc dword ptr [eax]pA.a   //Move pointer to A into eax
                                    //Increment it. Yes, this is correct.

    //pA->DoA();
    void (A::*pFunction)() = A::DoA; //Call non-virtual function
    A::DoA.                          //Declare 'pFunction' to be a pointer to
    __asm mov    ecx, dword ptr [pA] //Put the 'this' pointer into ecx (VC++-
specific)                           specific)
    __asm mov    edx, dword ptr [pFunction] //Put pointer to member function in edx
    __asm call   edx                  //Call member function

    //pA->DoAVirtual();
    __asm mov    eax,dword ptr [pA]   //Call virtual function
    __asm mov    edx,dword ptr [eax]  //Put pointer to vTable in eax
into edx                           //Put pointer to pointer to DoAVirtual
    __asm mov    ecx,dword ptr [pA]  //Put the 'this' pointer into ecx (VC++-
specific)                           specific)
    __asm call   dword ptr [edx]     //Call the function pointed to by edx
    address.
}

```

The C++ language standard actually specifies the "asm" keyword. It is used like this:

```
asm ( <compiler- and processor specific stuff here> );
```

Thus, this code would be legal on PCs:

```
asm ( mov eax, 1234 );
```

This standard is so new that it is just now appearing in current compilers; Code Warrior and Borland C++ Builder 3.0 are the only ones I know of that support it. I recommend that you stick with the older asm usage for now.

C++ Portability

This is a brief section that discusses what you need to know about writing portable C++ code.

The C++ language was finally standardized in November of 1997. But it's going to take at least a year or two before the major compiler vendors are complaint with the standard. The majority of the actual syntax of the language hasn't changed in the last 2 years. The bulk of the changes to the language since then have related to the standard C++ library. Since the standard C++ library is now fixed, public domain versions of the standard library will become available, accelerating the process of compliance by compiler vendors.

Writing portable C++ code is much like writing portable C code. But since the C++ language has only recently been standardized, you probably want to write more conservative code than the standard allows. By "conservative," I mean that you should write code that complies with the standard as of about 3 years ago. Simply refrain from using the more recent language syntax and library additions. Given the state of compilers for gaming platforms today, I offer these lists of things you can probably use safely and things you may not be able to use safely:

Very Safe	Moderately Safe	Unsafe
new/delete classes public/private/protected inheritance multiple inheritance virtual functions inline references templates casting operators (e.g. <code>const_cast</code>) nested classes operator overloading exception handling stream IO	STL (Standard Template Library) string class RTTI (Run Time Type Identification) bool namespaces	mutable explicit stringstreams localization support (locale/facet, etc.) asm export explicit valarray exotic syntax (e.g. forward declaration of nested templates)

I'm telling you right now: **resist the urge of Andy Academia to use all the latest features of the language**. Andy also likes to use exotic syntax, which is just asking for trouble. At Maxis, the features in the "Unsafe" column are off-limits. We had a guy once (no longer an employee) that wanted to start using `mutable`, since it became available with VC++ 5. He said, "we need to use mutable; it's critical." My response was, "That's funny, because we've gotten along fine for the last six years without it. Why is it suddenly critical." Simply put: write simple code.

Current commercial PC compilers tend to be more up to date than console and embedded system compilers. As soon as the GNU compiler is revised for the ANSI standard (probably will happen within the next year or two), all three columns will be quite safe.

Here's another bit of advice: don't do knee-jerk compiler upgrades. Just because Microsoft just posted the latest patch to VC++ doesn't mean you should have everybody stop what they're doing and install it. Have one willing person be the Guinea Pig. Let him or her test it for a week or two. Then, when you feel confident, have everybody upgrade at once, if possible. If the new version of the compiler offers nothing more than a few minor bug fixes, consider not upgrading at all. Programmers hate stopping their development to go through risky compiler upgrades. Imaging doing it 3 or 4 times during a project. Microsoft is already on the fourth version of VC++ 5 as you read this. This advice really has little to do with C++, but C++ programmers seem to be a little more inclined to do these kinds of upgrades.

Localization with C++

Localization of all large game development projects is mandatory these days. We're going to examine C++ library support for localization in the context of the Standard C++ Library, a custom C++ localization library, and Operating System localization services. At Maxis, localization is almost effortless from a programming perspective. This is because we have developed a set of libraries and conventions that make localization efforts very smooth. If you don't yet have such a smooth system, you will by the time you're done with this section.

In making an application that looks proper in other countries, a number of issues are common. At the core of this is character sets. The ASCII character set defines the look of 8 bit characters from 0-127. Extended-ASCII defines the look of all 256 8 bit characters. However, some languages need to use characters that aren't present in this set of 256. For these characters, we need different character sets. English, German, French, Italian, and others use the "Western European" character set, which is much like extended-ASCII. Czechoslovakia, Poland, and other use the "Eastern European" character set. Asian, Russian, Indian, and Greek languages use yet other character sets. So we see that for each character set, there are multiple languages that use it.

In addition to languages and character sets, there are character representation types. Western character sets can get by with 1 byte per character, because the total number of required characters is less or equal to 256. Some character sets, most notably the Asian character sets, need more than 256 characters. To deal with this, multi-byte and double-byte character sets were invented. Multi-byte character sets use one byte per character, but some characters are reserved and mean that the subsequent character is to be interpreted differently. Double-byte character sets use two bytes for each character. Since this means a total of 65536 possible characters, it is in theory possible to store *all* characters for *all* character sets in a single double-byte character set space. This what Unicode is based on.

Writing an application that transparently deals with all this seems daunting at first, but it is possible. Read on.

Win32 Localization Support

If you're writing Windows programs, you may be interested in Windows localization support. Windows provides pretty good localization support for Win32 applications, provided you don't mind doing things the Windows way. The primary mechanism for localization under Windows is MBCS (MultiByte Character System) and Unicode. Windows NT fully supports Unicode. Windows 95 has little support for Unicode. Windows 98 support for Unicode is unknown at this time, but it appears to be the same as Windows 95.

Here are the core Win32 localization support functions, with the first set relating to string manipulation and the second group relating to fonts and rasterization:

CharLower	CharLowerBuff	CharNext	CharNextExA
CharPrev	CharPrevExA	CharToOem	CharToOemBuff
CharUpper	CharUpperBuff	CompareString	ConvertDefaultLocale
EnumCalendarInfo	EnumCalendarInfoProc		EnumCodePagesProc
	EnumDateFormats		
EnumDateFormatsProc	EnumLocalesProc	EnumSystemCodePages	EnumSystemLocales
EnumTimeFormats	EnumTimeFormatsProc	FoldString	FormatMessage
GetACP	GetCPInfo	GetCurrencyFormat	GetDateFormat
GetLocaleInfo	GetNumberFormat	GetOEMCP	GetStringTypeA
GetStringTypeEx	GetStringTypeW	GetSystemDefaultLangID	GetSystemDefaultLCID
GetThreadLocale	GetTimeFormat	GetUserDefaultLangID	GetUserDefaultLCID
IsCharAlpha	IsCharAlphaNumeric	IsCharLower	IsCharUpper
IsDBCSLeadByte	IsTextUnicode	IsValidCodePage	IsValidLocale
LCMapString	LoadString	lstrcat	lstrcmp
lstrcmpi	lstrcpy	lstrcpyn	lstrlen
MultiByteToWideChar	OemToChar	OemToCharBuff	SetLocaleInfo
SetThreadLocale	WideCharToMultiByte	wsprintf	wvsprintf
AddFontResource	CreateFont	CreateFontIndirect	
	CreateScalableFontResource		
DrawText	DrawTextEx	EnumFontFamilies	EnumFontFamiliesEx
EnumFontFamExProc	EnumFontFamProc	EnumFonts	EnumFontsProc
ExtTextOut	GetAspectRatioFilterEx		GetCharABCWidths
	GetCharABCWidthsFloat		
GetCharacterPlacement		GetCharWidth	GetCharWidthFloat
	GetCharWidth32		
GetFontData	GetFontLanguageInfo	GetGlyphOutline	GetKerningPairs
GetOutlineTextMetrics		GetRasterizerCaps	GetTabbedTextExtent
	GetTextAlign		
GetTextCharacterExtra		GetTextColor	GetTextExtentExPoint
	GetTextExtentPoint		
GetTextExtentPoint32	GetTextFace	GetTextMetrics	PolyTextOut
RemoveFontResource	SetMapperFlags	SetTextAlign	SetTextCharacterExtra
SetTextColor	SetTextJustification	TabbedTextOut	TextOut

One problem with Windows text output is that it is done through GDI (Graphics Device Interface). If you are writing a high-performance DirectDraw game, the only way to draw onto a DirectDraw surface is by calling `GetDC ()`, doing text drawing with GDI, and then calling `ReleaseDC ()`. This is slow and not even guaranteed to work. You cannot do GDI operations onto a 3Dfx card surface, for example.

Also, Windows 95 unfortunately does not let you set the locale on the fly. Basically, if you use the Win32 localization functionality, the user must set the system locale manually and changing it requires a

reboot. This may not be a big deal for end-users, but it makes development and QA very tough. You want to be able to play the game in any language or locale by simply altering a setting, and possibly restarting the game. Lastly, Windows cannot provide support for a given locale unless the user (or your application) has specifically installed support for that locale.

In summary, using Win32 localization support provides the advantage of having already written and tested code, it otherwise brings a number of disadvantages for games programmers, such as slowness, potential unavailability, and non-portability.

C/C++ Locale Support

The first versions of both C and C++ had little support for localization. This has finally changed. Both C and C++ have ISO and ANSI standardized localization support. The C localization support is reasonably well documented in *The Standard C Library*, but P.J. Plauger. The C++ localization support is unfortunately not described very well in any current book other than the *C++ Language Standard* itself.

Here is a list of all the core localization-savvy C library functions provided by MSVC++. Other compilers have a similar set:

atof	atoi	atol	isalnum	iswalnum	islower	iswlower
isalpha	iswalpha	isprint	iswprint	__isascii	iswascii	ispunct
iswpunct	iscntrl	iswcntrl	isspace	iswspace	__iscsym	__iscsymf
isupper	iswupper	isdigit	iswdigit	isxdigit	iswxdigit	isgraph
iswgraph	iswctype	isleadbyte	localeconv	_mbccpy	_mbclen	mblen
_mbstrlen	mbstowcs	mbtowc	printf...	scanf...	setlocale	_wsetlocale
strcoll	wscoll	_stricoll	_wcsicoll	_strncoll	_wcsncoll	_strnicoll
_wcsnicoll	strftime	wcsftime	_strlwr	strtod	wcstod	strtol
wcstol	strtoul	wcstoul	_strupr	strxfrm	wcsxfrm	tolower
towlower	toupper	towupper	wcstombs	wctomb	wtoi	_wtol

Here is a list of the relevant C++ localization classes:

locale	facet	id	ctype
--------	-------	----	-------

Unfortunately, the C++ localization classes have only been recently added to the standard and aren't available in many C++ implementations yet. Since the C++ standard was ratified in 1997, it won't be long before these classes are universally supported. Nevertheless, the C++ localization classes suffer from one of the problems that Win32 suffers from: you only get localization support for the locales that come with the package. This may not be a problem if all you plan on supporting is the major European locales, though.

Custom Locale Support

The advantage of rolling your own localization support is that there are no limitations to what you can do or what platforms or locales you can support. The disadvantage is that you must write everything yourself. Well, we're going to cheat—I'm going to give you much of what you need to get started. The localization classes we provide here aren't academically perfect, but they will work for virtually any game you plan to localize.

If you want to make your app run under Unicode or your own double-byte system, you'll definitely want to abstract the character type. What you do is use `"Char"` instead of `"char"` wherever text is involved. Also, instead of using the C++ `string` class (uses single byte chars only) directly, you make a class called `String`. Below is a platform-independent version of the code you need to do this. Put this in a file called `"Char.h"` or something similar. At Maxis, we have used something just like this for a couple years now with good results. Note that you don't have to run under a Unicode-savvy operating system to use Unicode; you simply need to follow the Unicode 16 bit character conventions to make your application Unicode-savvy on any platform.

```
#ifndef _UNICODE
    typedef unsigned short Char;    //- Used wherever you previously used 'char'.
    typedef unsigned short Int;     //- Needed for Unicode-safe C/C++ run-time lib
    funcs.
    #define _EOF                    WEOF    //- EOF definition that works for both Unicode.
    #define Text(x)                 L ## x  //- Used on string literals, like this:
    typedef wstring                 String  //      Char* strPtr    = Text("Blah");
#else //ASCII
    //      Char character = Text('B');
    typedef char                   Char;    //
    typedef int                    Int;     //
    #define _EOF                    EOF     //
    #define Text(x)                 x       //
    typedef string                 String   //- Note that we abstract the string class too.
#endif
#ifdef _WINDOWS
    #define _EOL Text("\n\r")
#else
    #define _EOL Text("\n")
#endif
```

Next, you'll need a "Language Manager" C++ class. This class implements locale management for your application. With the Language Manager, you can query for supported locales, change the current locale, ask what character set should be used for a given locale, etc. Here is an example of a Language Manager much like the one we use at Maxis:

```
class LanguageManager{
public:
    enum LanguageType{
        kLanguageNone           = -1,
        kLanguageDefault        = 0,
        kLanguageBegin           = 1,
        kLanguageUSEnglish       = kLanguageBegin,
        kLanguageEnglish         = kLanguageUSEnglish,
        kLanguageUKEnglish,
        kLanguageFrench,
        kLanguageGerman,
        kLanguageItalian,
```

```
kLanguageSpanish,  
kLanguageDutch,  
kLanguageRussian,  
//Add new identifiers here  
kLanguageJapanese,  
kLanguageCount = kLanguageJapanese+1  
};
```

```

enum CharacterSet{ //Similar to the concept of a code page.
    kCharSetUnknown      =   -1,
    kCharSetDefault      =    0,
    kCharSetThai         =   874, //Thai
    kCharSetJapanese     =   932, //Japanese, double-byte (~8000 chars)
    kCharSetChineseRep   =   936, //Chinese (Rep. of China), double-byte (~8000
chars)
    kCharSetKorean       =   949, //Korean, double-byte (~18,000 chars)
    kCharSetChineseHK    =   950, //Chinese (Hong Kong), double-byte (~15,000
chars)
    kCharSetEasternEuropean = 1250, //Eastern European (~256 characters)
    kCharSetCyrillic      = 1251, //Bulgarian
    kCharSetWesternEuropean = 1252, //Western European (~256 characters)
    kCharSetGreek         = 1253, //Greek
    kCharSetTurkish       = 1254, //Turkish
    kCharSetHebrew        = 1255, //Hebrew
    kCharSetArabic        = 1256  //Arabic
};

enum RoadDrivingSide{
    kRoadDrivingSideLeft  = 0,
    kRoadDrivingSideRight
};

enum MeasurementSystem{
    kMeasurementSystemEnglish = 0,
    kMeasurementSystemMetric
};

public:
    //Standard Functions
    virtual bool    Init();
    virtual bool    Shutdown();
    virtual void    AddAvailableLanguage(int nLanguage);
    virtual void    RemoveAvailableLanguage(int nLanguage);
    virtual int     GetCurrentSystemLanguage();
    virtual bool    SetCurrentLanguage(int nLanguage = kLanguageDefault);
    virtual bool    CanSwitchToLanguage(int nLanguageToSwitchTo);
    virtual void    GetAllAvailableLanguagesIntoStringList(StringList& stringList);
    virtual bool    GetLanguageRuntimeLibraryName(String& sLanguage,
        int nLanguage = kLanguageDefault);
    virtual bool    GetNextAvailableLanguage(int& nStartingLanguage);
    virtual bool    GetLanguageDirectoryName(String& sLanguage,
        int nLanguage = kLanguageDefault)=0;
    virtual bool    GetLanguageEnglishName(String& sLanguage,
        int nLanguage = kLanguageDefault)=0;
    virtual bool    GetLanguageLocalName(String & sLanguage,
        int nLanguage = kLanguageDefault)=0;
    virtual bool    GetLanguageIDFromLanguageEnglishName(String& sLanguage, int&
nLanguage);
    virtual bool    GetLanguageIDFromLanguageLocalName(String& sLanguage, int&
nLanguage);
    virtual bool    GetLanguageEnglishNameFromLanguageID(String& sLanguage,
        int nLanguage = kLanguageDefault);
    virtual bool    GetLanguageLocalNameFromLanguageID(String& sLanguage,
        int nLanguage = kLanguageDefault);
    virtual int     GetLanguageIDAlias(int nLanguage = kLanguageDefault);
    virtual LanguageUtility* GetLanguageUtility(int nLanguage = kLanguageDefault);
};

```

Lastly, you'll need a "Language Utility" class. This class does all the locale-specific dirty work, like formatting text. Here is an example of a Language Utility class much like the one we use at Maxis:

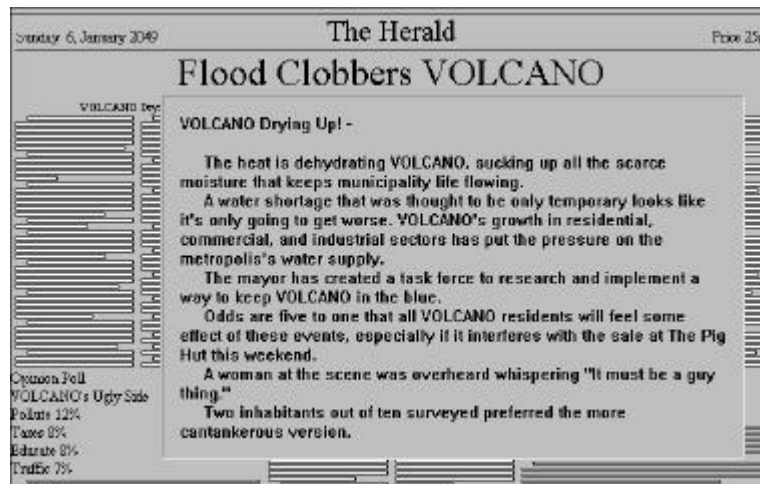
```
class LanguageUtility{
public:
    //Time/Date
    virtual bool GetDayName(int day, String& dayString); //Mon=1, Sun=7
    virtual bool GetAbbrDayName(int day, String& dayString);
    virtual bool GetMonthName(int month, String& monthString); //Jan=1
    virtual bool GetAbbrMonthName(int month, String& monthString);
    virtual bool MakeTimeString(int hour, int minute, int second, String& destString);
    virtual bool MakeDateString(int month, int day, int year, String& destString);

    //Numbers and money
    virtual bool GetCurrencySymbol(String& currencyString);
    virtual bool DoesCurrencySymbolPrecedeAmount();
    virtual bool IsSpaceBetweenCurrencySymbolAndAmount();
    virtual bool GetThousandSeparator(String& separator);
    virtual bool MakeMoneyString (int amount, String& destString);
    virtual bool MakeNumberString(int number, String& destString);

    //Character Set Utilities
    virtual void ConvertToLowerCase(String& string);
    virtual void ConvertToUpperCase(String& string);
    virtual void ConvertToLowerCase(Char& character);
    virtual void ConvertToUpperCase(Char& character);
    virtual bool IsCharLower      (Char& character);
    virtual bool IsCharUpper      (Char& character);
    virtual bool IsCharAlpha      (Char& character);
    virtual bool IsCharNumeric    (Char& character);
    virtual bool IsCharAlphaNumeric(Char& character);
    virtual bool IsCharPrintable  (Char& character);
    virtual bool CanCharEndLine   (Char& character);
    virtual bool CanCharStartLine (Char& character);
    virtual bool IsCharPunct      (Char& character);

    //Misc functionality
    virtual int  GetCharacterSetFromLanguage();
    virtual bool DoesLanguageUseWesternCharacterSet();
    virtual bool DoesLanguageUseEasternCharacterSet();
    virtual bool GetLanguageRoadDrivingSide();
    virtual int  GetMeasurementSystem();
};
```

In closing this section on localization, I'd like to offer one bit of advice. Don't try to write a program that attempts to implement locale-specific grammar on the fly. It is certainly possible to write a game with a user interface that simply displays strings for the given locale, and merely displays a different string for a different locale. And minimal formatting can even be done as well. However, if you try to form proper sentences dynamically, you're going to have all kinds of trouble and regret the attempt to do it. In SimCity 2000 (1994), we had a newspaper that would come up and display status information to the user. The idea was cool: use a newspaper motif to convey the status of the city to the user. Unfortunately, localizing the text and grammar in a generic way for this newspaper was hell. We won't be doing that again.



For SimCity 3000, we opted instead for a single line newsticker metaphor. This way, at most only a single independent sentence at a time had to be localized.



Game Frameworks in C++

One of the best uses of C++ is the implementation of application frameworks. By "application framework," we mean a library of code used to implement the common parts of an application, such as the main window, user interface, graphics primitives, user input, networking, and so on. MFC, OWL, and TCL are examples of these. But MFC is generally considered a bad idea for most games, especially given its general incompatibility with DirectX, its bulkiness, and its lack of cross-platform support. Besides, users want a game with a unique look and feel to it, a game that looks as much like a work of art as it is a playable game.

Nevertheless, C++ application frameworks are still a good idea for game development, but you'll probably have to write the framework yourself. One of the most important reasons to develop a framework, or even merely a 3D rasterization library, is *independence*.

Maybe you want to wrap DirectX with an easier-to-use interface. That's one of the things we have at Maxis; it's called DDD (which means 3D).

You may want to have one 3D interface for all platforms, with the interface being non-specific to any platform. That's one of the things we have at EA (parent of Maxis); it's called *3RASH*. It is a 3D rasterizer that works well on DirectX, PlayStation, N64, Macintosh, Glide, DOS, and potentially other platforms as well.

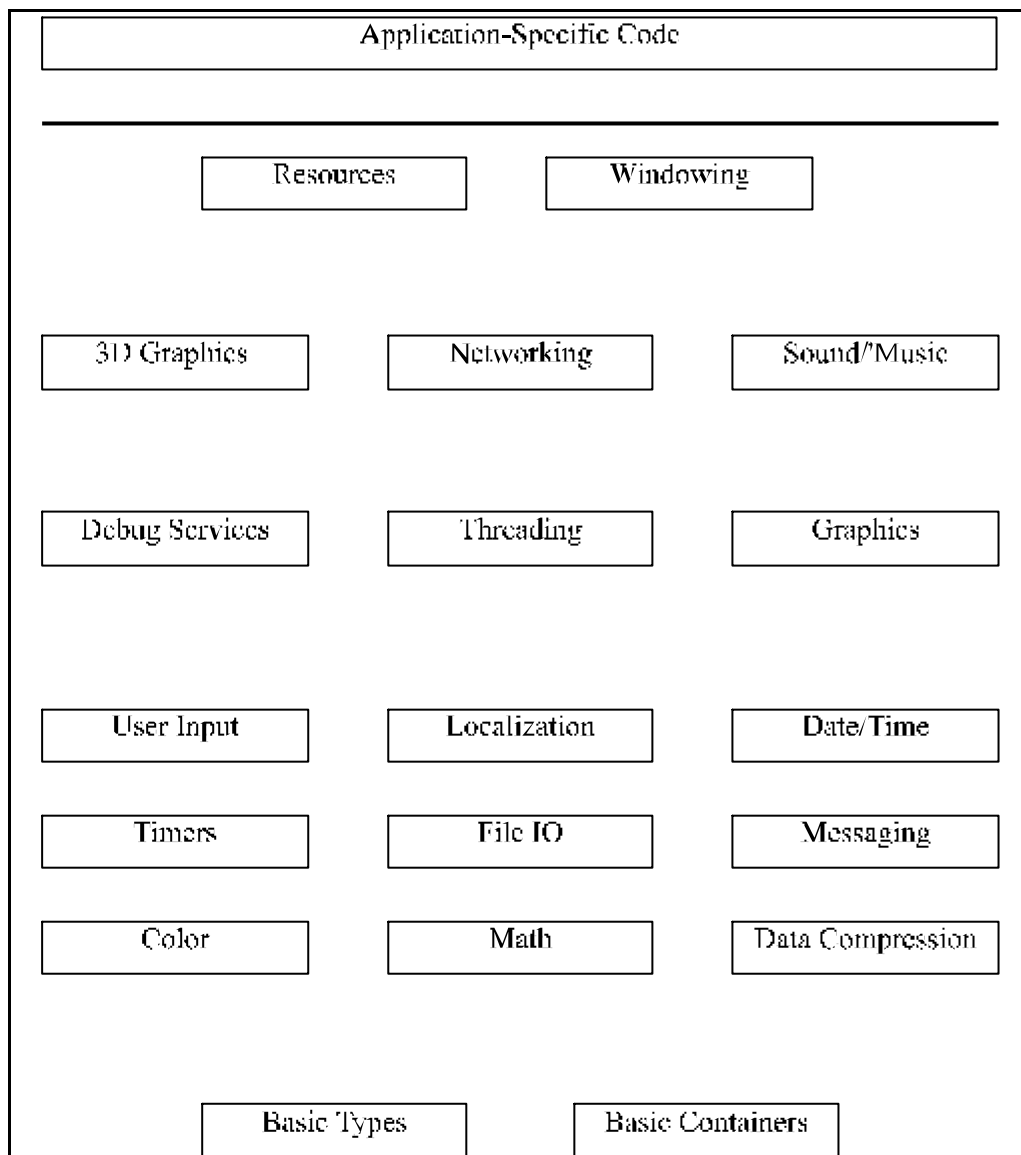
Maybe you want to have not just the 3D rasterization abstracted, but all the major common subsystems that a game can use, from 3D to UI to sound. That's one of the things we have at Maxis; it's called *The Framework*. Actually, the current Maxis framework has gone through a number of name changes and used to be comprised of parts called SPARKAL (SimPark Abstraction Layer), SWINDAL (SPARKAL Windowing Layer), SCOUNDRAL (SimCopter Sound Layer), Gonzo (Muppet), and Rizzo (Muppet). Now, we just call it *The Framework*.

What we're going to talk about here is how to go about successfully designing and implementing a C++ game application framework. There are some big lessons we've learned about this at Maxis, and we're going to touch on the most significant ones here.

But first, is it really feasible to write a generic C++ framework for games? Is C++ fast enough? Aren't games so unique that you can't possibly write generic code for any game? Yes. Yes. No. It's certainly feasible to write a generic C++ framework for games, because we've done it and we've seen others do it as well. C++ is certainly fast enough. C++ is just as fast as C, as long as you know what you're doing. We've talked about this to some degree already, and there is a detailed document on the topic in the 1998 CGDC Conference Proceedings. And while the best games have their own distinctive look and feel, the fact is that underneath that facade is a body of code that could be used to implement another distinctive game.

On the next page is a diagram of an example framework.

Example Framework Components



There are a couple important aspects of the diagram on the previous page:

- Not all the modules mentioned above are required, and others that aren't shown above are also possible. This should go without saying.
- Each part of the framework is a separate module usually identified by a class or set of classes.
- The application-specific code sits on top of everything and the application-specific code is invisible to the framework. This is critical to a successful framework implementation.
- Each module is dependent only on modules in layers below it, and not modules above it. This is "layering" and it is critical to a successful framework implementation.

Here are some important things to know about this type of architecture:

- Even with the layering mentioned above, modules should still be made as independent as possible.
- Make core low-level classes featherweight, totally public, and without virtual functions. For example, if you want to define a 2D point class, define it like this:

```
struct 2DPoint{
    int x;
    int y;
};
```

instead of like this:

```
class 2DPoint{
public:
    int GetX() return x; //This provides *no*
    int GetY() return y; //benefit to anybody.
protected:
    int x;
    int y;
};
```

If you want to make a fancy feature-laden 2DPoint class, make it a subclass of the basic 2DPoint. Trust me, if your base class is complicated, people aren't going to want to use it.

- Absolutely, make your resource loading/management system independent of the rest of your framework. The resource system can use the framework, but the framework has no idea the resource system works. The resource system is visible to only the application-level code.
- Andy Academia likes this kind of layered organization. Just make sure he doesn't get carried away with over-layering, which is another problem in itself. We had a small problem with this early in our framework development with some parts of it. Old Andy Academia took what could have been one or two classes and made them into eight classes. We spent a couple days cleaning up that mess.
- Try to find a balance between C++ canonical form and lightweight classes. Recall that canonical form means creating classes with the following members: default constructor, copy constructor, `operator=()`, `operator==()`, and so on. While this makes classes very container-friendly, it's also annoying to write for all classes and will lead to code bloat when overused.

- When possible, provide both high- and low-level interfaces to the classes. For example, when designing a serious sound system, it's nice to have a high-level Load/Play interface; it will get used 90% of the time. However, you'll also need a low-level direct buffer manipulation interface; it will be useful for when you want to write a special sound manipulation engine or something like that.
- When in doubt, follow common design patterns. See the **C++ Resources** section.
- You're going to run into the problem of version control and compatibility. Team A on project A starts in January and ships in December. Team B on project B starts in June and ships the next February. Well, what's going to happen is that in December, Team A is going to discover that they cannot ship unless they make some small but significant change or bug fix in the framework. So they make the change. But Team B cannot incorporate Team A's critical change, because it would break something in their project. The framework has diverged. This will happen, and it's a problem that really has little to do with C++. I can say that the enforced modularity of a C++ framework will make this problem easier to deal with. I can't say that there is an easy solution or way to prevent the problem.
- It's a good idea to have some sort of consistent naming scheme in your framework, but I wouldn't try too hard on this. I guarantee you that if you try to come up with one consistent perfect naming scheme for everything, some day somebody is going to add some class from some other project or something that doesn't use your naming scheme.
- Try to hide platform-specific code behind platform-independent classes, rather than make platform-specific classes. We discuss this biggie in detail below.

Platform-specific code

Suppose you want to write a `CallbackTimer` class. This class lets you specify a function that gets called after so many milliseconds. But unlike the `Rectangle` class, which is entirely platform-independent, the `CallbackTimer` class is best implemented using the low-level timer services of the operating system or hardware. What you want to do is make a platform-independent header file (`CallbackTimer.h`) whose (public) interface is independent of any operating system. But your `CallbackTimer.cpp` file makes all the OS-specific calls. You make a separate version of `CallbackTimer.cpp` for every supported platform. What you usually *don't* want to do is make separate classes for every platform, like this:

```
class CallbackTimer
class CallbackTimerWin32      : public CallbackTimer
class CallbackTimerMac        : public CallbackTimer
class CallbackTimerPlayStation : public CallbackTimer
```

If you do this, you are going to have two problems. First of all, you won't be able to subclass them! Here is a major C++ paradigm to always remember: C++ inheritance is *linear*. Subclassing works in only one direction. If you choose to make subclasses build upon parent classes by adding platform-ness, you're stuck with that direction. An example may help explain this. If you want to make a subclass of `CallbackTimer` called `PeriodicCallbackTimer`, and you've done the above platform-specific subclassing thing, you've got a big mess on your hands. The best you can do is create multiple `PeriodicCallbackTimer` classes, one for each platform, like this:

```
class PeriodicCallbackTimerWin32 : public CallbackTimerWin32
class PeriodicCallbackTimerMac   : public CallbackTimerMac
class PeriodicCallbackTimerPlayStation : public CallbackTimerPlayStation
```

This sucks. None of it would be necessary if you stayed away from using the inheritance mechanism to impart platform-specific capabilities to classes.

Secondly, users won't be able to instantiate a `CallbackTimer` without doing one of two things:

- Putting `#ifdefs` everywhere, like this:

```
#if defined(WIN32) //This method is horrible
    pTimer = new CallbackTimerWin32;
#elif defined(MAC)
    pTimer = new CallbackTimerMac;
#elif ...
```
- Implementing a factory system to create `CallbackTimers` instead of creating them with `new()` or as automatic variables. We talk about factories a bit in the next section. In summary, factories are powerful, but you'll want to use them only where needed, not for doing simple things like creating a little utility object. It's not that using factories is *wrong* for this application, but the developers will hate a framework that makes them jump through hoops to do simple things.

Factories

Factories are objects that make other objects for you. Instead of you writing code like this:

```
NetworkSocket* pSocket = new NetworkSocket;
```

A factory would have you write code like this:

```
NetworkSocket* pSocket = MakeNetworkSocket();
```

If you had a generic factory system, then it might look more like this:

```
NetworkSocket* pSocket = pGenericFactory->MakeObject(kNetworkSocketType);
```

Factories are useful for cases when you simply can't create the object with `new`. Imagine you have code in a Windows DLL or Macintosh Code Fragment, and you want the main executable to be able to create objects for the Code Fragment. But the Code Fragment doesn't have the source code for the class; it just has the header or a pure-virtual interface declaration. With a factory system, the Code Fragment need only have the factory make the object for it.

Examples

We now provide some examples of classes in this kind of framework. Full source code to the examples is available. First we start with the basic types header:

```
////////////////////////////////////
// BasicTypes.h

//Types
typedef unsigned   char    UInt8;
typedef signed     char    Sint8;
typedef unsigned   short   UInt16;
typedef signed     short   Sint16;
typedef unsigned   long    UInt32;
typedef signed     long    Sint32;

// Limits
const UInt8  MAX_UINT8      = (UInt8)      0xFF;
const Sint8  MIN_SINT8      = (Sint8)      0x81;
const Sint8  MAX_SINT8      = (Sint8)      0x7F;
const UInt16 MAX_UINT16     = (UInt16)     0xFFFF;
const Sint16 MIN_SINT16     = (Sint16)     0x8001;
const Sint16 MAX_SINT16     = (Sint16)     0x7FFF;
const UInt32 MAX_UINT32     = (UInt32)     0xFFFFFFFF;
const Sint32 MIN_SINT32     = (Sint32)     0x80000001;
const Sint32 MAX_SINT32     = (Sint32)     0x7FFFFFFF;
const Float32 SMALLEST_FLOAT32 = (Float32) 1.175494351e-38F;
const Float32 BIGGEST_FLOAT32  = (Float32) 3.402823466e+38F;
const Float32 MIN_FLOAT32      = (Float32) -BIGGEST_FLOAT32;
const Float32 MAX_FLOAT32      = (Float32) BIGGEST_FLOAT32;
const Float64 SMALLEST_FLOAT64 = (Float64) 2.2250738585072014e-308;
const Float64 BIGGEST_FLOAT64  = (Float64) 1.7976931348623158e+308;
const Float64 MIN_FLOAT64      = (Float64) -BIGGEST_FLOAT64;
const Float64 MAX_FLOAT64      = (Float64) BIGGEST_FLOAT64;
////////////////////////////////////
```

Now here's an example of a nice templated rectangle class:

```
////////////////////////////////////
// Rect2D -- 2D rectangle
//
template <class T>
struct Rect2D{
    T left;
    T top;
    T right;
    T bottom;

    // Constructors
    Rect2D();
    Rect2D(T l, T t, T r, T b);
    Rect2D(const Rect2D& srcRect);

    // Operators
    Rect2D& operator= (const Rect2D& rect);
    bool operator==(const Rect2D& rect) const;
    bool operator!=(const Rect2D& rect) const;
    Rect2D& operator+=(const Rect2D& rect);
    Rect2D& operator-=(const Rect2D& rect);
    Rect2D& operator&=(const Rect2D& rect);
    Rect2D& operator|=(const Rect2D& rect);
    Rect2D operator+ (const Rect2D& rect) const;
    Rect2D operator- (const Rect2D& rect) const;
    Rect2D operator& (const Rect2D& rect) const;
    Rect2D operator| (const Rect2D& rect) const;

    // Queries
    T Width() const;
    T Height() const;
    bool IsRectEmpty() const;
    bool IsRectNull() const;
    bool PtInRect(T x, T y) const;
    bool PtInRectExclusive(T x, T y) const;
    bool DoesRectOverlap(const Rect2D& rect);
    bool DoesRectOverlapExclusive(const Rect2D& rect);

    // Operations
    void SetRect(T l, T t, T r, T b);
    void SetRect(const Rect2D& rect);
    void SetRectEmpty();
    bool EqualRect(const Rect2D& rect) const;

    void InflateRect(T h, T v);
    void InflateRect(const Rect2D& rect);
    void InflateRect(T l, T t, T r, T b);
    void DeflateRect(T x, T y);
    void DeflateRect(const Rect2D& rect);
    void DeflateRect(T l, T t, T r, T b);

    void MoveRect(T x, T y);
    void OffsetRect(T x, T y);
    void NormalizeRect(); //Ensures that the rectangle's left < right, top < bottom

    // Operations that fill the rect with result
    bool IntersectRect (const Rect2D& rect1, const Rect2D& rect2);
    bool IntersectRectExclusive(const Rect2D& rect1, const Rect2D& rect2);
    //For rectangles where you mean the right to be one pixel past the actual right.
    bool UnionRect (const Rect2D& rect1, const Rect2D& rect2);
    //Note that UnionRect works the same whether your rect is exclusive or not.
};
////////////////////////////////////
```

Now here's an example of a file IO class:

```
////////////////////////////////////
// class MFile
//
// Class to read/write files. We have avoided issues such as byte alignments,
// Unicode, and C++ string class issues here. So if you want to use this class,
// you might want to make small alterations to it. This file doesn't do text
// mode IO either. At Maxis, we have a slightly different version of this
// class that deals with these issues.
//
// Very simple example usage:
// char buffer[1024];
// MFile file("C:\\blah.txt");
// file.Open();
// file.Read(buffer, 1024);
// file.Close();
//
class MFile{
public:
    enum SeekMethod{           // Seek methods.
        kSeekMethodStart,     // Seek offset from beginning of file
        kSeekMethodCurrent,   // Seek offset from current position in file
        kSeekMethodEnd        // Seek offset from end of file
    };

    enum OpenMode{             // Open mode
        kOpenModeQueryAttributes = 0x0000, // Query attributes only
        kOpenModeRead            = 0x0001, // Open for reading
        kOpenModeWrite           = 0x0002, // Open for writing
        kOpenModeReadWrite       = 0x0003  // Open for reading and writing
    };

    enum OpenType{             // Open type
        kOpenTypeCreateNew,     // Fails if file already exists
        kOpenTypeCreateAlways,  // Never fails, always creates and truncates to 0
        kOpenTypeOpenExisting,  // Fails if file doesn't exist, keeps contents
        kOpenTypeOpenAlways,    // Never fails, creates if doesn't exist, keeps contents
        kOpenTypeTruncateExisting // Fails if file doesn't exist, truncates to 0
    };

    enum ShareMode{            // Sharing mode
        kShareModeNone        = 0x0000, // No sharing
        kShareModeRead        = 0x0001, // Allow sharing for reading
        kShareModeWrite       = 0x0002, // Allow sharing for writing
        kShareModeDelete      = 0x0004  // Allow sharing for deletion
    };

public:
    MFile();
    MFile(const char* szPath);
    virtual ~MFile();

    virtual bool IsOpen() const;
    virtual bool Open(const char* szPath = NULL, //If null, then we use the member variable path.
                     int openMode           = kOpenModeRead,
                     int openType           = kOpenTypeOpenExisting,
                     int shareMode          = kShareModeRead);
    virtual bool Close();

    // Position manipulation
    virtual int  GetPosition() const;
    virtual int  GetLength() const;
    virtual bool SetLength(int nNewLength);
    virtual int  SeekToBegin();
    virtual int  SeekToEnd();
    virtual int  SeekToRelativePosition(int nRelativePosition);
    virtual int  SeekToPosition(int nPosition);
    virtual int  Seek(int offset, SeekMethod origin); //Old-fashioned seek.

    // I/O
    virtual bool Read          (void* buffer,          unsigned long numBytes);
```

```
virtual bool ReadWithCount (void* buffer,      unsigned long& numBytes);  
virtual bool Write          (const void* buffer, unsigned long  numBytes);  
virtual bool WriteWithCount(const void* buffer, unsigned long& numBytes);  
virtual bool Flush();
```

```

// Other
virtual bool Remove();
virtual bool Rename(const char* szNewPathName);
virtual bool Exists() const;
virtual bool CopyFromFile(const char* szSourcePath, bool bOverwriteIfAlreadyPresent = true);

// Data accessors
virtual void      GetPath(char* szPath) const;
virtual const char* GetPath() const;
virtual void      SetPath(const char* szPath);

public:
    // Static functions - Functions that operate on files in general
    static bool  FileCreate(const char* szPath, bool bTruncateContents = false);
    static bool  FileExists(const char* szPath);
    static bool  IsFileWritable(const char* szPath);
    static bool  Remove(const char* szPath);
    static bool  Rename(const char* szOldPath, const char* szNewPath);
    static bool  Copy(const char* szSourcePath, const char* szDestPath,
                      bool bOverwriteIfAlreadyPresent = true);
    static bool  MakeTempPathName(char* szPath);
    static bool  MakeTempFileName(const char* szDirectory, char* szName);
    static int   Checksum(const char* szPath);
    static int   GetFreeDiskSpace(const char* szPath);
    static bool  GetPathExtension(const char* szPath, char* szExtension);
    static bool  GetPathDirectory(const char* szPath, char* szDirectory);
    static bool  GetPathName(const char* szPath, char* szName, bool bIncludeExtensionInName=true);
    static bool  GetPathDrive(const char* szPath, char* szDrive);

protected:
    char      szFilePath[_MAX_PATH]; // Full path to file
    bool      bOpen;                 // Whether or not the file is open
    void*      pPlatformSpecificData; // We would store a Windows hFile here.
};
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

Framework Conclusions

Let me tell you that a well-implemented framework will be very popular with your development team. The better the system is architected, the easier it will be to extend. New programmers will get up to speed very fast. Substituting different implementations of classes will be easy too.

Plug-in Systems in C++

Games can be made to be extensible with the use of "plug-in" technology. A simple example would be a case where a company like Blizzard wants to be able to add new types of weapons to Diablo II after shipping. A more involved case would be where a company like Maxis allows users to add *their own* buildings to SimCity 3000 after shipping. Implementing a plug-in architecture is a lot nicer in C++, due to various aspects of the language design that were intended specifically for such uses. I'm talking about pure-virtual interfaces. Here is an example of a pure-virtual C++ interface:

```
struct IDiabloWeapon{           //Notice that we intentionally make it a struct.
    This is
    bool Init()                  = 0;    //because structs are nothing but public classes.
    bool Shutdown()              = 0;
    int  GetType()               = 0;
    int  GetWeight()             = 0;
    bool GetBitmap(Bitmap**) = 0;
    ...
};
```

This interface would be the base class for all weapons. Here is a simplified example of a subclass:

```
class GurkaKnife : public IDiabloWeapon{
public:
    bool Init();
    bool Shutdown();
    int  GetType() { return 1001; }
    int  GetWeight() { return 100; }
    bool GetBitmap(Bitmap** bitmapPtrPtr) { return *bitmapPtrPtr = gurkaKnifeBitmap; }
    ...
protected:
    static Bitmap* gurkaKnifeBitmap;    //We only need one bmp for all instances of
    this class.
    static int      nBitmapsUsageCount; //We can keep a reference count of them here.
};
```

The game engine need not know anything about GurkaKnives. It only needs to work with IDiabloWeapons, just as long as you make the pure virtual interface complete enough. Here are some facts about pure-virtual interfaces as applied to plug-in systems:

- A pure virtual interface declaration takes up no run-time memory by itself. No vTable will be created or linked for it. Only when a concrete subclass is declared will such a vTable get instantiated.
- You want to make all functions pure-virtual, not just some of them. There is an issue with destructors – see below.
- Constructors are not necessary, since there is no data in the pure-virtual class.
- Destructors are useful to have in a pure-virtual class, though not necessary. If you add a destructor, however, you are unfortunately forced to making it *non*-pure-virtual—compilers usually fail on compile or link when you use pure-virtual destructors. This may or may not be much of an issue for you. Read on.

- Since pure-virtual destructors are effectively not possible, and since the actual concrete object subclassing from the interface may be in another DLL or code fragment, you may very well want to have an explicit `Delete()` function for the pure virtual class. It's important that your plug-in system provide a way to delete objects other than through `C++ delete`. This is because the object behind the pure-virtual interface may be in another DLL and may have even been written by another compiler. Thus, only the object can know how to delete itself. See below for a little more discussion of `Delete()` and `Release()`. You can't call `delete` in your main executable on a pointer that was allocated by anything (e.g. a DLL) outside your main executable.
- You can declare default function arguments in pure-virtual interfaces.
- Chances are that the concrete objects behind the pure-virtual interfaces can be from different versions of the same compiler or even from different compilers altogether. This is due to the ubiquity of the well-established virtual function table mechanism.
- Whatever system you end up implementing, you'll probably want to assign interface IDs to pure-virtual interfaces. Basically, every pure-virtual interface has an `int` that is an interface ID. You can then ask an object to give you a pointer to the give interface by simply supplying that object with an interface ID. You usually define that interface ID in the same header file as the pure-virtual class declaration.
- You may notice that since you are making interface functions pure-virtual, you are largely taking away the possibility for them to become inlined by the compiler. If a class is calling those same functions internally a lot, then the code can become inefficient. This is a fact of life, and would occur with or without C++ if you are trying to do practically any plug-in system. But there is a resolution: simply make two versions of the function that you'll want to call a lot, one virtual (the interface function) and one not virtual, but inline. Have the virtual one call the inline one. This doesn't happen so often that it becomes much of a problem.
- Be careful about passing parameters in registers with plug-in systems. This will have the effect of making it difficult or impossible to write a plug-in with a different compiler or language, since those other systems may not allow register passing the way that your compiler does. You may even have problems with future versions of your own compiler. Under PC compilers, the `__fastcall` declaration currently uses the `ecx` and `edx` registers to store the first two `DWORD` (32 bit) parameters. All others are pushed right to left. This may be incompatible with other compilers or languages, so be careful about using it.
- You probably can't rely on using C++ RTTI on a plug-in object pointer. At least not if it is possible that the object is in a DLL and was compiled by another compiler or another version of your own compiler. The RTTI mechanism is an internal compiler-dependent mechanism that is subject to change with compiler versions, and certainly has some differences between different compiler vendors.

Microsoft COM

Microsoft's version of plug-inability is called COM (Common Object Model). There are both C and C++ versions of it, but C++ is given preferential treatment. With the C++ version, pure-virtual interfaces are employed much as we have described them above. In COM, every C++ class is a subclass of a class called "IUnknown." Here is IUnknown:

```
class IUnknown{
    virtual ULONG    AddRef()    = 0;
    virtual ULONG    Release() = 0;
    virtual HRESULT QueryInterface(REFIID iid, void** ppvObject) = 0;
};
```

AddRef and Release implement a reference counting system on the object. Basically, right after you create an object with new, you call AddRef on it. AddRef then increments the reference count to one. If you then wanted to get rid of the object, you would call Release on it. Release would decrement the reference count, notice that it is now zero, and call "delete this."

QueryInterface is a system whereby the class can support inheritance, multiple inheritance, and composition without explicit use of these language features. Basically, QueryInterface allows for straightforward language-independent run-time type identification. Here is an example of a subclass of IUnknown:

```
class Demo : public IUnknown{
    enum { kDemoInterface = 12345678 };
public:
    Demo() : ref_count(0){ }
    ULONG AddRef(){
        return ++ref_count;
    }
    ULONG Release(){
        if(ref_count > 1)
            return --ref_count;
        delete this;
        return 0;
    }
    HRESULT QueryInterface(REFIID iid, void** ppvObject){ //REFIID is a 16 byte struct
(ugly)
        switch(iid){
            case kDemoInterface:
                *ppvObject = static_cast<Demo*>(this);
                AddRef(); //QueryInterface must always call AddRef under COM
                return 0; //'0' means OK in COM
            case kInterfaceUnknown :
                *ppvObject = static_cast<IUnknown*>(this);
                AddRef();
                return 0;
        }
        return E_NOINTERFACE;
    }
protected:
    int ref_count;
};
```

Reasons you may want to use COM instead of rolling your own plug-in system:

- The mechanism is well-defined and standardized by Microsoft
- The mechanism is *virtually* platform independent.
- If you are writing PC or possibly Macintosh games or tools, they may easily interface with tools and controls from other vendors who provide COM interfaces to their libraries or DLLs.
- The support code is largely already written and debugged.

Reasons you may not want to use COM, and instead might want to invent your own plug-in system:

- The COM reference counting system sounds elegant but in fact is annoying to use in big projects. We have found that there are continuous problems with memory leaks and unreleased objects on application shutdown. If all of the C++ language natively used such an `AddRef/Release` system, this wouldn't happen. But in practice, you can't know for sure if a pointer to a class is using reference counting or simple `new/delete`. I sometimes would be nice to have a simple `Delete()` call, which simply deletes the object rather than messes with reference counting. Remember, you can't call C++ `delete` on a pure-virtual plug-in object pointer. This is because the object may be in another DLL, and your heap manager won't know anything about it.
- There's a lot to know about Microsoft COM to implement it completely. The fact that there is a 400 page book devoted to COM is a testament to this. Game programmers don't like complicated or obfuscated libraries, especially those from Microsoft. The industry's disdain for Direct3D is a testament to this.
- You may not want to become dependent on something that is controlled and maintained by someone else. Game programmers are used to writing many things from scratch because they must support these things on numerous platforms.
- Microsoft COM REFIIDs (and COM GUIDs) are an ugly 16 byte struct, which would waste space and CPU cycles in most games. All you really need for most game development is a simple `int`.

Plug-Ins at Maxis

At Maxis, we took a route that is partly between rolling our own plug-in system and using Microsoft COM. We didn't want to become dependent on Windows nor the Microsoft-owned COM specification, but we thought it would be a good idea to implement something conceptually similar. So we came up with "GZCOM" (Gonzo COM). Here are the basic traits of GZCOM:

- Instead of MS COM's `IUnknown` base class, we implemented a nearly identical `IGZUnknown` class:

```
class IGZUnknown{
    unsigned long AddRef();
    unsigned long Release();
    bool QueryInterface(GZID id, void** ppInterface); // GZID is an int.
}
```
- We implemented our own version of COM's `GetClassObject`.
- Instead of using 16 byte COM GUIDs and REFIIDs, we use a simple 32 bit integer for IDs.
- We cut away most of the more esoteric COM features, such as Distributed COM and some of the DLL-interfacing functions.
- We implemented a simple DLL loading and interface-querying system, so anybody could easily write a DLL that provided new objects that the main executable could load easily and treat like native objects.
- Our version (GZCOM) is entirely implemented in just a few platform-independent and lightweight files.

Lessons we've learned after implementing GZCOM (to some degree, this represents my opinion rather than the aggregate opinion of the programmers at Maxis):

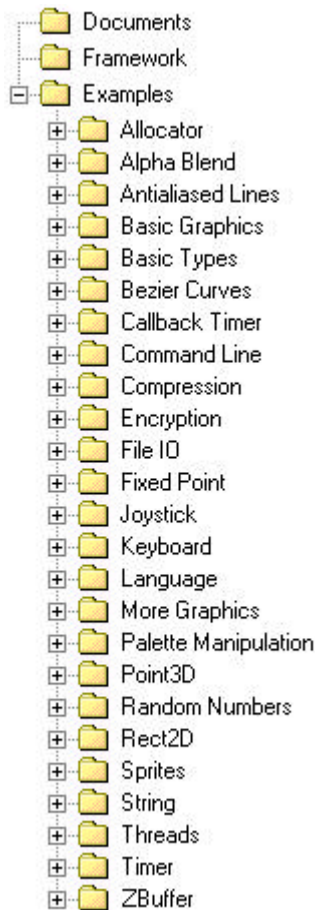
- Implementing GZCOM via standard C++ pure-virtual classes was a great idea. The mechanism is lightweight, reliable, and easy to learn.
- The `AddRef/Release` reference counting system was a mixed blessing. While it has allowed us implement reference counting on objects that truly need reference counting, such as shared 3D meshes, it has at the same time caused a lot of reference counting and leakage problems, due to reasons cited earlier.
- `QueryInterface` is an OK system for run-time type identification and safe casting. The fact that every call to it increments the reference count has become annoying. In retrospect, we should have added a boolean third argument to `QueryInterface`:

```
bool QueryInterface(GZID id, void** ppInterface, bool bAddRef=false);
```

There are some technical reasons why a plug-in system would want to increment a reference upon doing a `QueryInterface`, but those reasons tend not to affect game developers. Also, it would be a lot nicer in many cases to simply `static_cast` an interface pointer to the appropriate class rather than call `QueryInterface`. In fact, you can usually do this, as long as the necessary header files are visible. From a purely academic standpoint, incrementing the reference count on `QueryInterface` makes sense. From a usability standpoint, it's an annoyance.

- Watch out for Andy Academia and Patty Plug-In. Andy will complain when you use `static_cast` to get around the inefficiencies of `QueryInterface`, and Patty Plug-In will want to insist that practically every struct and class in the game be made plug-innable.
- This proliferation of plug-in interfaces is a real problem. I'm telling you right now that it's going to be very hard to resist some people's efforts to try to make virtually every class in the game plug-innable. This leads to programmers feeling like they are wading in molasses. You'll start to dread having to make new classes because of all the pure-virtual interfaces, interface IDs, and interface limitations you'll have.
- Speaking of interface limitations, you generally can't declare an interface for a class that takes an STL container as an argument if you want DLLs to be able to access the interface. This is simply because the DLL may be using a different version of STL or not understand STL at all. However, if you want, you can provide an STL version of an interface (for convenience within the exe) and a non-STL version (for external entities). Of course, you don't want code bloat either...
- You probably don't want to make every class in your game be pure-virtual and require that use of that class always go through the pure-virtual interface. One reason is that some classes really need intimate knowledge of each other's data structures for efficient operation.
- When writing classes that you want to be plug-innable, write a non-plug-innable version first. Get it working and debugged, then simply copy the public interface to another header file and make it pure-virtual. This will make development much easier.

The Distribution Disk



The distribution disk comes with a wealth of C++ code and 25 example programs. There are three directories: Documents, Framework, and Examples. The Documents directory has a number of documents, such as this file, the C++ draft standard and the High Performance C++ paper.

The Framework directory contains the entire source code to the Mini Framework. ***It is very important to understand that just because there are 70 files in the framework directory, you don't need all 70 to write an application.*** In fact, you can write an application with just two or three of them. The files in the framework are *modular* – each can be added as needed.

The Examples directory is a directory of example code using the framework. Each example demonstrates the usage of one part of the framework. The programs are small – usually the entire application is implemented in a single function: `main()`. The examples come with a VC5 project file. By the time you are reading this, Borland (Inprise) and Metrowerks project files may also have become available. If not, the projects are so simple that it would be very easy to get them running under any compiler.

The Mini Framework

The Mini Framework is high performance C++ framework for making platform- and locale-independent games. This framework is amazingly easy to use, yet is powerful and efficient enough to write virtually any game with, both 2D and 3D. All games at Maxis, including the recently shipped SimCity 3000, are developed with a framework almost identical in architecture to this one. SimCity 3000 has shipped nearly simultaneously for three platforms and 10 languages, including Far East languages and has already broken the Electronic Arts record for the biggest initial release for a PC product. So you can believe me when I tell you this kind of application framework works for demanding commercial games.

No 3D?

I'm going to tell you right off the bat that there is no 3D support in the Mini Framework. At Maxis, we do have a 3D module that plugs into the framework, but it is proprietary. The Mini Framework is a base upon which you would add your own 3D system. The fact is that if I was to supply a 3D system here, it probably for one reason or another would not suit your specific needs, and you'd need to replace it or significantly modify it. Such is the nature of 3D graphics. I do plan on adding support for OpenGL some day, and when I do, it will be available to you.

Here are some of the features of the framework:

- High performance 8, 16, 24, and 32 bit basic graphics engine.
- Built-in ability to run in a desktop window (for easy debugging) or full-screen.
- Built-in ability to run on any graphics card (2D or 3D) in any supported resolution.
- Basic graphics primitives with clipped and non-clipped versions of each: blit, stretch blit, line draw, set/get pixel, fill, etc.
- Advanced graphics primitives with clipped and non-clipped versions of each: anti-aliased lines, alpha blending, flood fill, etc.
- Built-in localization support.
- Multi-threading services, including threads, fast mutexes and critical sections.
- Fast good random number generator class (generates integers and floating point).
- Fast, powerful data encryption (via the TwoFish algorithm).
- Easy lossless data compression.
- Date and Time classes.
- User input support: keyboard, mouse, joystick.
- Color conversion help: Convert 24 bit to 16 bit, 555 to 565, RGB to HLS/CYMK, etc.
- Command line parsing class. Ask it if "-d" is present and it returns yes or no, and more.
- Easy file IO class.
- Code to load bitmap files into game surfaces and textures.
- Sound playing/looping.
- Stopwatch timers (for accurate timings).
- Callback timers (for calling you at a regular time interval).
- Cool fixed point math class.

- Bezier curves galore.
- Super Fast memory allocation.

Here is a diagram of the classes represented in the framework as of this writing (2/5/99):

(* Denotes classes that may or may not make it into the final distribution disk)

Layer 3	<div>Resources</div> <div>Sound*</div>		
Layer 2	<div>Debug Services</div>	<div>Multithreading</div>	<div>Graphics</div>
Layer 1	<div>Command Line</div>	<div>Callback Timers</div>	<div>Utilities</div>
	<div>Random Numbers</div>	<div>Math (fixed point)</div>	<div>Memory Allocation (fast!)</div>
	<div>User Input</div>	<div>Localization</div>	<div>Date/Time</div>
	<div>Stopwatch Timers</div>	<div>File IO</div>	<div>Data Encryption</div>
	<div>Color</div>	<div>Messaging</div>	<div>Data Compression</div>
Layer 0	<div>Basic Types</div> <div>Basic Containers</div>		

For most of the classes, there is a header file (.h) and a source file (.cpp). An example of this is Compression.h and Compression.cpp. Some of the files are simply lone header files (.h). An example of this is Keys.h.

On the next few pages we will go through some FAQs and examples of how to use the framework. In actuality, most modules in the framework comes with example code at the top of the header file. Each of the examples is on the distribution disk in the Examples directory.

How Do I ... ?

Here we answer some common questions you may have about the mini framework. It can do a lot of things, but these things need to be listed and explained. So we present this section in the form of a FAQ to answer these questions. The next section provides actual examples

Q. How do I set the resolution the game runs in. Ditto for bit depth?

A. The `GraphicsManager` has a function called `PreInitSetDesiredGameResolution` and a function called `SetScreenResolution` that let you set the desired resolution before game start and during game runtime respectively. If the app is running in windowed mode, you may not be able to change the screen resolution.

Q. I want to run my game app in windowed mode so I can debug it. How do I do this?

A. The debug build of the app runs in windowed mode by default, but you can always force the behaviour to windowed or full screen by calling the `GraphicsManager` function `PreInitSetFullScreenMode`.

Q. Why does the app insist on running in full-screen mode when I explicitly called `GraphicsManager.PreInitSetFullScreenMode(true)`?

A. Because the bit depth you requested for the app was different from the one the screen is already in. While this behaviour is platform-dependent, under DirectX and Win32 you cannot reliably change the screen bit depth or resolution unless you run in full screen exclusive mode. The fix for this is to change you bit depth to match the one the game runs in.

Q. How do I run the game with double-buffering or triple-buffering?

A. You call the `Graphics Manager` function `PreInitSetBacksurfaceCount`. Passing 1 as an argument (default) specifies double-buffering. Passing 2 as an argument specifies triple-buffering.

Q. How do I get the app to run on a different card from the primary, such as my 3Dfx?

A. You call the `GraphicsManager` function `PreInitSetDriverIdentifier`. This function takes as an argument a character array. This character array can either be a string that appears in the driver name, or it can be a unique numerical identifier assigned to the driver. Under DirectX, this numerical identifier is the driver GUID. For example, you can run on your 3Dfx by calling:

```
char array[kVideoDriverIDLength] = "3Dfx";  
GraphicsManager.PreInitSetDriverIdentifier(array);
```

Q. What does `GraphicsManager::PreInitSetScreenSurfaceMemoryType` do?

A. This function is another `GraphicsManager` "PreInit" function that controls the graphics system initialization. If you specify system memory with this function, then the back buffer is put into system memory instead of video memory. It only has any effect if you are running in windowed or non-flipping mode; in full-screen flipping mode, the primariy surface and the back buffer must both be in video memory.

Q. When `CallbackTimer` calls my callback function, what thread is it in?

A. Technically speaking, this is platform-dependent. Under Win32, your callback function will be called in a separate thread from the main thread. This thread is created by the callback timer system. So watch out for thread safety issues.

- Q. I get crashes or hangs only in full-screen mode with the debug version of the app. What's happening?
- A. Under DirectX, if you hit a breakpoint or the app brings up an assert, the system will hang. That's DirectX.
- Q. How do I use surface locks?
- A. Class `Surface` has a function called `Lock` which locks the surface for pixel access. As such, it acts like the DirectX surface locking mechanism. As such, you'll want to call `Lock` before calling `SetPixel` or `SurfaceBits`. However, class `Surface` also has a `DrawLine` function, which, under DirectX, will require the surface to be locked to operate. In practice, the `DrawLine` function will lock the surface internally if it is not already locked. However, for efficiency, you can lock the surface ahead of time if you know you will need to do many `DrawLine` calls. Ditto for other primitives that need pixel access. This is a hard concept to implement in a platform-independent way.
- Q. How do I lock the front surface of the screen surface?
- A. When you call `Lock` on an offscreen (standard) surface, it locks that surface for bit access. `ScreenSurface` is a subclass of `Surface` which has two parts – the front surface and the back surface. Normally, operations on a screen surface (such as `Lock`) operate on the back surface, since that is usually what games want to do. However, you can call `ScreenSurface::SetMainDrawSurface` to change this behaviour. For example, if you call `ScreenSurface.SetMainDrawSurface(ScreenSurface::kMainDrawSurfaceFront)`, then all subsequent operations happen to the front (video monitor) surface.
- Q. Some graphics primitives have no effect when I run the app in some bit depths. What is happening?
- A. Not all graphics primitives have been implemented in all bit depths (4, 8, 16, 24, 32 bpp). You'll notice that the source code to these functions have a comment to this effect. You can take the time to implement them if you need them badly enough. 16bpp is the most supported format for graphics primitives.
- Q. I've used some of the alpha blending functions and they generate junk on the screen. Why?
- A. You must call `Init8BitAlphaSystem`, `Init16BitAlphaSystem`, `Init24BitAlphaSystem`, or `Init32BitAlphaSystem` before using the alpha blending functions. These init functions set up blending tables as needed for the operations. See the example code for usage and examples. As of this writing (2/99), only the 16 bit alpha initialization function has been completed.
- Q. Can I subclass or substitute `Surface`, `ScreenBuffer`, `MainWindow`, or `GraphicsManager`?
- A. These are the four classes central to a basic graphics application. You can in fact either subclass or entirely replace any of these classes with a new implementation, provided that your subclass or new implementation supports the same interface as the one in the framework. Notice that `GraphicsManager` has a function called `SetScreenSurface`. If you call this function with your own `ScreenSurface` before calling `GraphicsManager::Init`, then the `GraphicsManager` will use your screen surface instead of an internally created one.
- Q. What's the story with all the unimplemented functions in the file `SurfaceFunctions.h`?
- A. These are function declarations for some advanced primitives that haven't been defined as of this writing (2/99). Implementing these functions would take more time than was available. Please feel free to implement any of these and submit them for use by others.
- Q. How do I get cursor and keyboard events from the app or system?
- A. The `MainWindow` processes these messages and will optionally pass them on to you if you call `MainWindow::SetEventMessageTarget`. See the `BezierCurves` example for usage of both keyboard and cursor events. See `MainWindow.h` for an enumeration and description of all the event types.

Q. How do I make a `Surface` from a disk `.bmp` file?

A. The header file `BitmapLoad.h` defines functions to do this. Currently, support is limited to Win32 `.bmp` files, though, as with the rest of the framework, the functions declarations themselves are platform independent and thus would allow you to write an implementation for any bitmap type. In commercial games, bitmaps are usually compressed in a big dat file, but that's outside the scope of what we are concerned with.

Q. How random is the random number generator class `Random`?

A. The `Random` class generates random numbers that are good enough for 98% of commercial games, including simulations. Perhaps only commercial gambling games require more robust generators. The design of the `Random` class was motivated by four factors: speed, small size, 32 bit numbers, and good randomness. The `C rand` function generates only 16 bit random numbers, forces exactly one per application executable, and doesn't provide a good way to properly set the range. The `Random` class addresses these issues and more, yet is roughly as fast as the `C rand` function.

Q. How do I use multithreading classes (and compression, z-buffers, encryption, random numbers, timers, etc.)?

A. See the example code. There are examples for almost all the major components of the framework.

Q. How secure and efficient is the data encryption provided by class `Encryption`?

A. The `Encryption` class is based on the TwoFish encryption algorithm, and is powerful enough that as of this writing it is one of the leading candidates to be the successor to DES as the government encryption standard. It is faster than DES and yet more secure than DES. The assembly version on a Pentium II machine is on the order of 20 clock ticks per encrypted byte. The C version is maybe half as fast. This should be fast enough for most uses, especially since most of this encrypted data is for sending over a network, which is slow in comparison.

Q. Are the `Date` and `Time` classes Y2K compliant?

A. A rigorously accurate answer to this question may require an expert on the topic, but on the surface there appear to be no such issues. In particular, classes `MDate` and `MTime` do not store or use years as only 2 digit entries, they are always used as 4 digit years. If `MDate` is given a year as an argument that is less than 100, then it converts the year to 1900+year and stores the full four digit year.

Q. I swear `MTimer` is wrong and is mis-timing my code. Is this possible?

A. You can be assured that `MTimer` almost certainly is not broken. It often happens that people are surprised at how low or high a timing result is and come to the conclusion that `MTimer` is broken. Let me tell you now that `MTimer` has been used a 100 times under 100 different circumstances and has not been shown to be wrong yet.

Q. If the `BasicTypes.h` header defines a number of types, like `UInt32`, why don't the majority of the framework files use these types?

A. Experience has shown that it is best to separate modules from each other as much as possible. Since most of the framework modules don't really need to have the argument types so rigorously defined, there is no need to use these types. A framework in which you need the entire framework to be present just to compile one file is going to be a major pain to work with. An example of this is MFC, which for all practical purposes is completely non-modular – you must have virtually all of the MFC headers present to simply compile one of the `cpp` files. Not so with the Mini Framework. While classes like `GraphicsManager` and `ScreenSurface` will require as many as five or six other framework headers (small number in itself), classes like `MTimer`, `MDate`, `MRandom`, `MCompression` and others require no framework headers at all – they stand entirely on their own.

Basic Graphics

```
#include <GraphicsManager.h>
#include <MainWindow.h>
#include <ScreenSurface.h>
#include <MessageLoop.h>

void main(){
    GraphicsManager graphicsManager;
    MainWindow      mainWindow;
    ScreenSurface*  pScreenSurface;

    graphicsManager.PreInitSetDesiredGameResolution(ScreenResolution(800, 600,
16));
    mainWindow.SetGraphicsManager(&graphicsManager);
    mainWindow.SetTitle("Basic Graphics Application");
    if(mainWindow.Init()){
        if(graphicsManager.Init()){
            pScreenSurface = graphicsManager.GetScreenSurface();
            mainWindow.SetScreenSurface(pScreenSurface);
            while(MessageLoop()){
                //Draw game on ScreenSurface here.
                pScreenSurface->Swap();
            }
            graphicsManager.Shutdown();
        }
        mainWindow.Shutdown();
    }
}
```

This is about the smallest full graphics application you can make with the framework. Four header files are needed: GraphicsManager, MainWindow, ScreenSurface, and MessageLoop. A basic description of each of these is in order. Here is a small table that does this:

GraphicsManager	This class manages the game's graphic system. It initializes and maintains the basic graphics settings and environment. It lets you pick the game resolution and bit depth, what video card you are running on, and whether you want a double-buffered flipping surface or simply a single-buffered blitting surface, etc.
ScreenSurface	This class represents the screen drawing surface itself. It is important to differentiate this from the GraphicsManager, for reasons we'll see later. The screen surface can be flipped, blitted to/from, filled, locked for pixel access, etc. ScreenSurface is in fact a subclass of the Surface class, which itself represents a generic surface.
MainWindow	This class represents the ScreenSurface viewport. While the ScreenSurface represents the visible color display, the MainWindow represents the input into that display. The ScreenSurface and MainWindow are closely related but they are often better made as separate entities.
MessageLoop	This is not a class but a single function that processes input messages and operating system messages that the application may want to know about.

The first thing the app does is create a `GraphicsManager` and `MainWindow`. You don't need to create a `ScreenSurface` because the `GraphicsManager` will create a default one for you if you don't create one yourself. The app then tells the `GraphicsManager` that it would like the application to run in 800x600x16 bit color by calling `PreInitSetDesiredGameResolution`. Then the `MainWindow` is given a pointer to the `GraphicsManager` by calling `SetGraphicsManager`. Then `MainWindow::Init()` is called and `GraphicsManager::Init()` is called. Lastly, we ask the `GraphicsManager` for a pointer to the `ScreenBuffer` it created and hand that pointer to the `MainWindow`. So until now we've simply created these three entities and made sure they all know about each other. We can subclass *any* of these entities and provide overloaded behaviour if desired, all without modifying the original classes. Now the app is set up.

Now that the app is set up, we simply repeatedly draw on the `ScreenSurface` and call `Swap`. When we're done, we simply shutdown the objects in the reverse order of how we initialized them.

Notice how easy that is – seven simple function calls to completely set up the system. Anyone who's programmed `DirectDraw` knows that it would take hundreds of lines to accomplish this same thing. In fact, there are many powerful options that the `GraphicsManager` and `ScreenSurface` provide that aren't shown above. Check out the header file or ask me if you are interested.

More Graphics

Now that we've created a basic empty graphics application, let's add to it by implementing an application that uses some of the graphics features of the framework. The setup of the application is nearly the same as with the Basic Graphics application, so we show mostly only the message loop portion below. See the More Graphics example in the Examples directory for full source.

```
#include <Bezier.h>
#include <BitmapLoad.h>
#include <AlphaBlend.h>
#include <AntialiasedLine.h>

Rect2D  rect1;
Point2D points[1000];
int     color1;
Surface* pSprite;

Init16BitAlphaSystem(graphicsManager.GetScreenPixelFormat());
pSprite = SurfaceResourceLoader::SurfaceFromFile("Sprite.bmp", 16,
                                                Surface::kMemoryVRAMLowPriority);
pSprite->SetTransparentColor(pSprite->ConvertRGBValueToNative16Bit(255, 0, 255));

while(MessageLoop()){
    //Blank out the screen.
    pScreenSurface->Fill(0);

    //Draw a sprite at a random position.
    pSprite->BltTo(pScreenSurface, &Rect2D(0, 0, pSprite->Width(),
                                           pSprite->Height()), &Point2D(rand()%640, rand()%480));

    //Make a random filled rectangle of a random color.
    rect1.left  = rand()%640;          rect1.top   = rand()%480;
    rect1.right = rect1.left+(20+rand()%300); rect1.bottom = rect1.top +(20+rand()%300);
    pScreenSurface->Fill(&rect1, 15+rand()%32000);

    //Draw a random rectangle outline of a random color.
    rect1.left  = rand()%640;          rect1.top   = rand()%480;
    rect1.right = rect1.left+(20+rand()%300); rect1.bottom = rect1.top +(20+rand()%300);
    pScreenSurface->DrawRectangle(&rect1, 15+rand()%32000);

    //Draw a random line of a random color
    pScreenSurface->DrawLine(rand()%640, rand()%480, rand()%640,
                             rand()%480, 15+rand()%32000);

    //Draw a random anti-aliased line of a random color.
    ::DrawLineAntialiased(pScreenSurface, rand()%640, rand()%480,
                           rand()%640, rand()%480, 15+rand()%32000);

    //Draw a random Bezier curve.
    if(pScreenSurface->Lock()){
        Point2D pt1(rand()%640,rand()%480), pt2(rand()%640,rand()%480),
                pt3(rand()%640,rand()%480), pt4(rand()%640,rand()%480);
        Bezier2D4Controls(pt1, pt2, pt3, pt4, points, 1000);
        for(int i=0; i<1000; i++)
            pScreenSurface->SetPixel(points[i].x, points[i].y, color1);
        pScreenSurface->Unlock();
    }

    //Set some random pixels to random colors.
    if(pScreenSurface->Lock()){
        for(int i=0; i<500; i++)
            pScreenSurface->SetPixel(rand()%640, rand()%480, 15+rand()%32000);
        pScreenSurface->Unlock();
    }

    pScreenSurface->Swap();
}
```

Note how easy it is to do the operations above. Most of the graphics operations can be done in a single line of code, yet these graphics are very fast – fast enough for most commercial games. You can find more graphics code on the distribution disk.

Compression

The Compression class is a simple class that implements lossless data compression and decompression. As of this writing (2/14/99), the class uses the well-known ZLib compression scheme. The interface is very easy to use. Here it is:

```
class Compression{
public:
    Compression();
    ~Compression();

    bool    CompressData    (const void* pDataIn, int nDataLengthIn,
                           char*      pDataOut, int& nDataLengthOut);
    bool    DecompressData (const char* pDataIn, int nDataLengthIn,
                           void*      pDataOut, int& nDataLengthOut);
    int     GetLengthOfCompressedData    (const char* pDataIn);
    int     GetLengthOfDecompressedData (const char* pDataIn);
    int     GetMaxLengthRequiredForCompressedData (int nLengthDataInput);
};
```

That's all there is to it. You simply compress data with `CompressData` and you decompress that data with `DecompressData`. You don't need to know or care how the data is compressed nor how it is stored, the class does this for you. If you have a chunk of compressed data, and want to know how big that data is when uncompressed, call `GetLengthOfDecompressedData` with the compressed data as a argument. If you have a chunk of compressed data and want to know how big this compressed chunk is, call `GetLengthOfCompressedData`. The `CompressData` function also returns this value in the `nDataLengthOut` argument. Below is an example function that tests the Compression class.

```
bool TestCompression(){
    Compression compression;
    const int    kBytes(10000);
    char         pOriginalDecompressedData[kBytes],
                pDecompressedData[kBytes], pCompressedData[kBytes*2];
    int          nDecompressedDataSize(kBytes), nCompressedDataSize(kBytes*2);
    int          i, nErrorCount(0);

    for(i=0; i<kBytes; i++)
        pOriginalDecompressedData[i] = pDecompressedData[i] = (char)i/4;
    for(i=0; i<kBytes*2; i++)
        pCompressedData[i] = -1;

    if(compression.CompressData(pOriginalDecompressedData, nDecompressedDataSize,
                               pCompressedData, nCompressedDataSize))
    {
        if(compression.DecompressData(pCompressedData, nCompressedDataSize,
                                      pDecompressedData, nDecompressedDataSize))
        {
            if(nDecompressedDataSize == kBytes){ //Now verify that everything went
OK.
                for(int i=0; i<kBytes; i++){
                    if(pOriginalDecompressedData[i] != pDecompressedData[i])
                        nErrorCount++;
                }
                return nErrorCount == 0;
            }
        }
    }
```

```
    }  
  }  
  return false;  return false;  
}
```

Random Numbers

The Random class is a nice random number generator for games. It generates both integer and floating point values, and generates a few different distributions as well (i.e. not just even distributions). The C runtime library function `rand()` is a bad random number generator if anything simply for the reason that it only generates 16 bit random numbers (0-65535). The Random class generates true 32 bit random numbers and generates them as fast or faster than C's `rand()` function. Additionally, generating a random number between 0 and 50000 with C `rand()` by simply taking `rand()%50000` will actually generate a non-random distribution!

There is a lot of academic theory about random numbers out there, and it is interesting reading. An expert will be quick to tell you that nobody has ever seen a perfect software random number generator. Some of the best random number generators use arrays and tables of values as seeds, not just a single integer. The Random class presented below uses the simple integer seed approach, yet generates *very good* random numbers for virtually all game purposes with probably the lone exception of true government regulated Las Vegas casino games. The reason we take the simple seed approach is that it makes creating and seeding a single random number generator object very fast. The result of this is that you can create them on the fly, if desired.

Here is the public interface:

```
class Random{
    Random(int nSeed=-1);
    void Seed(int nSeed=-1);

    bool      RandomBool();                //Returns true or false;
    unsigned RandomUIntUniform();          //0 to limit-1
inclusively,                               // uniform distribution
    unsigned RandomUIntUniform(unsigned limit); //0 to limit-1
inclusively,                               // uniform distribution
    int      RandomIntRangeUniform(int start, int end); //start to end-1
inclusively,                               // uniform distribution
    int      RandomIntRangeGaussian(int start, int end); //start to end-1
inclusively,                               // normal (Gaussian)
    distrib.

    double RandomDoubleUniform();          //[0.0 to 1.0),
                                           // uniform
    distrib.
    double RandomDoubleRangeUniform(double start, double end); //[start to end),
                                           // uniform
    distrib.
    double RandomDoubleGaussian();         //[0 to 1), Normal
                                           // (Gaussian)
    distrib.
    double RandomDoubleRangeGaussian(double start, double end); //[start to end),
    Normal
                                           // (Gaussian)
    distrib.
};
```

Here is some example usage of this class:

```
void TestRandom(){
    Random random; //Will seed self based on system clock.

    int randomIntegerBetween0and1000000    = random.RandomIntUniform(1000000);
    int randomIntegerBetweenMinus50and2000 = random.RandomIntRangeUniform(-50,
2000);
    int randomIntegerBellShapedDistrib      = random.RandomIntRangeGaussian(0, 100);

    double randomDoubleBetween0and1         = random.RandomDoubleUniform();
    double randomDoubleBetween34and500      = random.RandomDoubleRangeUniform(34.0,
500.0);
    double randomDoubleBellShapedDistrib    = random.RandomDoubleRangeGaussian(-1.0,
1.0);
}
```

Now that's easy and powerful. In fact, you can use the `RandomUIntUniform` function to as a base for other non-uniform distributions, if you like.

File IO

The File class is very nifty indeed. If you are familiar with the MFC CFile class or the Borland TFile class, then you'll be comfortable with this class. In fact, the File class we present is better than the others because the interface is platform independent and it has more features. We have already seen the header file earlier in this document, so we won't repeat it here. We will give some example code, however, since this class is so useful.

```
void TestFileIO(){
    char  buffer[1024];

    MFile file("C:\\\\blah.txt");           //Here we read 1024 bytes of a
    file.Open();                           //  file into a buffer
    file.Read(buffer, 1024);
    file.Close();

    file.Open("C:\\\\blah2.txt",           //Here we open a file for writing,
              MFile::kOpenModeWrite,      //  creating it if it didn't exist
              MFile::kOpenTypeOpenAlways); //  already.
    file.Write("abcdefg", 7);             //Here we write seven bytes to the
file
    file.Close();

    file.Open("C:\\\\blah3.txt");          //Here we open a file for reading and
    file.Read(buffer, 1024);              //  do some operations on it.
    file.SeekToPosition(20);              //Go to position 20 in the file.
    file.SeekToRelativePosition(5);       //Now go to position 25 in the file.
    int size = file.GetLength();           //Get file length.
    file.SetLength(300);                  //Set file length to 300.
    file.Close();

    bool bResult;                         //Here we test some of the cool
statics.
    bResult = File::FileExists("C:\\\\a.txt"); //See if this file exists.
    File::Rename("C:\\\\a.txt", "C:\\\\b.txt"); //Rename a file.
    File::Remove("C:\\\\b.txt");           //Delete a file.
    File::MakeTempPathName(char* pPath);   //Have it create a temp file path.
    int size = File::GetFreeDiskSpace("C:\\\\"); //Get free disk space on C.
    File::GetPathName("C:\\\\b.txt", pPath); //Extract the file name from a
path.
}
```

Note that we call the class MFile instead of File. The standard for Mini (as in Mini Framework). Some of the Mini-Framework files and classes start with an M because they have a decent chance of colliding with other class names you may have, because the names are so common.

Threads

Threads are very useful for high performance applications, and they have definite uses for games. SimCity 3000 for Windows runs in four threads: Main thread, simulation thread, music streaming thread, and background graphics loader thread. In practice, you'll want to be careful about how you use threads in your application – the discussion of this is beyond the scope of this book.

What we provide in the Mini Framework are classes and functions for basic threading support. The actual thread context switching code is usually part of the operating system, so we simply have our classes call OS functions where needed. Some operating systems don't have true pre-emptive multi-threading support, so you'll want to consider the platforms you'll be running on before using multi-threading. Often, you can design your app so that it runs with multi-threading off or on. In fact, SimCity 3000's three extra threads can all be disabled, making the app run in a single thread. This results in jerky performance, so we prefer the use of threads where possible.

The Mini Framework provides the following threading support

- Threads
- Fast Locks Quick thread-safe boolean lock.
- Critical Sections Quick thread-safe re-entrant lock.
- Mutexes Interprocess version of Critical Section
- Semaphores Counted lock

Additional classes are possible. At Maxis, we have these and a few others, such as Signal, MutexSet, and others.

Now we present some code to show how to use these classes. If you aren't familiar with the concepts of threads or threading, you'll want to read about that elsewhere.

```
unsigned ThreadFunction(void* pData){
    bool* bShouldQuit = (bool*)pData;
    while(!*bShouldQuit)
        ; //Do something here.
}

MThread thread;
bool bShouldQuit(false);

thread.SetPriority(MThread::priorityBelow3); //Set thread to an initial low
priority.
thread.Begin(ThreadFunction, &bShouldQuit); //Create a thread. Pass address as
pData.
MThread::SleepCurrentThread(1000); //Sleep the current thread for 1
second.
thread.SetPriority(MThread::priorityAbove1); //Set thread to a high priority.
thread.Suspend(); //Suspend thread.
thread.Resume(); //Resume thread.
bShouldQuit = true; //Send msg to thread to quit.
Thread.Join(); //Wait for the thread to exit.
```

C++ Resources

If you want to do C++ development, there are a number of important standard resources that you'll want to know about. Here is a list of such resources and some comments on each. While a number of the references are related to Microsoft and Visual C++, this is not because I am trying to push that product. It is simply because companies big and small tend to try to align themselves with Microsoft, resulting in a lot of resources related to Visual C++.

Books

Source	Comments
The C++ Programming Language, 3 rd edition, by Bjarne Stroustrup	Buy this book. No questions asked. Stroustrup is an excellent author who is very interested in efficient code. Practically anything you'll come across in C++ is covered in this book, with example code.
The C++ Standard ftp.research.att.com/dist/c++std/WP/CD2/ www.maths.warwick.ac.uk/cpp/pub/wp/html/cd2/	This is the C++ (draft) standard document itself. The final approved standard is not yet available for the public yet, due to various legal/rights issues. Believe it or not, it's actually very good reading. It's not very difficult, but I wouldn't read it until you have a decent grasp of the language, because this document covers the entire gamut of the language. Available in draft form as a PDF document at the ftp address and in a frame-based searchable web site at the www address. The PDF version of this document is on the distribution CD for this class.
Effective C++/More Effective C++ by Scott Meyers	Useful information about C++ and how to use it. Also spends some time talking about the internals of C++ code generation. Effective C++ (2 nd edition) says on page 17 to throw away <code>printf/scanf</code> and embrace C++ streaming IO. Let me tell you that that sentence was <i>not</i> meant for game programmers.
C++ FAQs by Cline and Lomow	<p>Great book to get you on the road to being an advanced C++ programmer. Since it is separated into easily-digestible chunks, you don't have to read it from front to back. One issue I have with it is that it doesn't answer a lot of questions that I personally would consider frequent. You get the impression that the authors made up the list of "frequently asked questions" themselves, rather than actually finding out what users really want. For example, high on the list of truly frequently asked questions is "How do I overload operator <code>new()</code>?" The book doesn't address this question.</p> <p>There is an interesting discussion in the book on whether a circle is properly a subclass of an ellipse. Check it out. I personally agree with their basic argument (that it is not a proper subclass), but in practice, I would still consider subclassing a circle from an ellipse <i>simply because it is convenient to do so</i>. C++ is a tool, not a religion.</p>
The Design and Evolution of C++ by Bjarne Stroustrup	This book is interesting because it includes a rationale and history for virtually every significant feature or non-existent feature in the language. Read page 290 to see why they opted not to support the <code>inherited</code> keyword.



C++ Primer by Stanley Lippman	This is the standard book to start with, if you're just beginning in C++. It concentrates more on the language than OO design and architecture. You'll get good at that with time. No matter how great a C programmer and architect you are, you'll need some time before you can really crank with C++. Sorry, that's just the way it is. But trust me, you'll be glad you did it.
Advanced C++ Idioms by James Coplien	Interesting book for academics; less useful for games programmers. It's one of those books that "broadens your horizons" by showing you the unusual or unexpected ways the C++ language can be used. Unfortunately, some game programmers sometimes take these ideas to an extreme and start writing obfuscated and inefficient code. Be on the watch for this.
The Standard C library by P.J.Plauger	This book covers the C language standard, as opposed to C++. Nevertheless, C++ programmers often resort to using the C library for a variety of tasks, such as the printf/scanf functions mentioned above. This book covers everything and has a decent section on C localization. If you want to understand what it takes to write truly portable and localizable code, this book is the ticket. Games programmers probably aren't so concerned with this, however.
Essential COM by Don Box	Good book about Microsoft COM. Believe it or not, it's probably a good idea to understand COM and other competing plug-in interface systems (like the Java system) before embarking on writing your own.
Design Patterns	This is a classic book about common architectural or implementation issues in programming. It's not specific to C++, but you'll swear after reading it that the authors had object-oriented programming in mind when they wrote it.
Death March by Edward Yourdon	Good book about managing big projects. And these days, game development is getting big.
Code Complete by Steve McConnell	A good book about how to do programming in the real world. Includes coding tips, debugging tricks, and advice about project organization. Unless you are writing code like it's done in this book (and at Maxis, we do), you are probably not being as efficient as you could be.
Developing International Software by Nadine Kano.	This is not a C++ book, but it is referenced in this document. While this is a Windows programming book, it actually doubles quite well as a general book and reference for anyone interested in writing localized applications. It covers virtually everything you need to know, and things you really didn't want to know (read: things to get worried about). It also has neat character set and keyboard references in the appendices.

Periodicals

The C++ Report www.sigs.com/publications/cppr/	Good C/C++ magazine.
C/C++ Users Journal	Good C/C++ magazine.

www.cuj.com	
Dr. Dobbs Journal www.ddj.com	<p>Good generic programming magazine which uses C/C++ as its primary language. Often covers topics of indirect interest to game programmers.</p> <p>There was a recent article in DDJ that discussed how to do very efficient matrix math with C++: "Scientific Computing: C++ Versus FORTRAN" by T. Veldhuizen, Nov. 1997. C programmers who deride C++ often like to use matrix math as an example of how C++ is "inefficient" compared to C. In fact, when done properly, C++ is <i>at least</i> as efficient as C with matrix math.</p>
Windows Developer's Journal www.wdj.com	<p>Not just Windows. It is largely applicable to all platforms. Has a section called "Bug++ of the month," where they dissect some compiler's C++ bug every issue. Those bugs always get fixed by the compiler vendor, so if you have a definite bug you think your compiler vendor is ignoring, submit it to Bug++ (wdletter@mfi.com).</p>
Game Developer	<p>Most of you are familiar with this magazine. It doesn't cover language issues specifically, though it did have an interesting article in 1997 that described the (lack of) optimization smarts of various current compilers. Result: Intel compiler optimizes best, but hand-optimization beats all.</p> <p>There was an interesting article by Brian Hook in the Jan. '98 issue where he talks about C and C++ programming on page 14. Unfortunately, his argument is a little naive. About the language, he says "It's constantly evolving, getting bigger and uglier, and pretty soon it's going to implode under its own weight." This is his basic argument. Actually, what was getting bigger and uglier was the standard library, not the language itself. 97% of the language, and 100% of the important parts of the language, have been set in stone for years. You don't have to use a single part of the standard library if you don't want to, and for game programming we usually don't use the standard library.</p>
MSJ (Microsoft Systems Journal) www.msj.com	<p>Thus used to be a great Windows programming resource. It seems to have become a bit too driven by Microsoft's marketing and political agenda in the last couple years (in my opinion).</p> <p>It has a section called "C++ QA", but this is a lie: it's an MFC section only, and never covers C++ itself. This MFC section is always funny to read, at least if you're somewhat anti-MFC. This is because practically every month this section features a question posed by a user that asks why some really simple thing can't be done with MFC. The columnist proceeds to then explain about how MFC has this and that quirk and bug and the only way to do what you want involves this painful process that takes an entire column to explain. It's a bit ironic, because the columnist is Paul DiLascia, author of <i>Windows++</i>, a book that invented an excellent Windows application framework before MFC even existed.</p>

Internet Resources

news://comp.lang.c++.	<p>Like many other news topics, there is a high noise-to-signal ratio here. People spend a lot of time complaining about bugs in their compilers, debating the differences between Borland, Microsoft, and Watcom, and flaming back and forth about C vs. C++.</p> <p>Bjarne Stroustrup (the inventor of C++) occasionally responds to threads and recently (February, '99) responded three times in a thread about the relative efficiency of C and C++. One of those three comments was a slight correction of a statement that I made (oops!). I stated that the C++ 'asm' keyword was relatively new to the standard, when in fact it has always been there, according to Bjarne.</p> <p>Nevertheless, you can still get basic or straightforward questions answered if you title your subject appropriately readers are feeling generous. I personally have a policy of answering at least one question for every one I post.</p>
news://comp.lang.c++.moderated	<p>A slightly more academic crowd hangs out here. If you post anything to this newsgroup, it <i>must</i> be a generic compiler-independent C++ question, or it will be rejected. The signal to noise ratio here is better than with comp.lang.c++, but your chances of getting an answer are also smaller.</p>
news://msnews.microsoft.com/microsoft.public.vc.*	<p>Microsoft news server root for VC. At least it doesn't have all the spam that you find on regular news servers. Microsoft employees apparently do not actively monitor these newsgroups.</p>
news://msnews.microsoft.com/microsoft.public.vc.language	<p>Microsoft news server for C++ language. The name sounds like it deals with the language, but guess what? 70% of the messages are about other things, like why the CListCtrl doesn't work right...</p>
www.microsoft.com/VISUALC/	<p>Microsoft's Visual C++ page.</p>
www.microsoft.com/kb/	<p>Microsoft C++ technical support page. When dealing with Microsoft technical support, expect to be treated like a bonehead and expect to have them try to send you useless documents that have nothing to do with your problem. I'm sorry Microsoft, that is simply my experience.</p>
loki.borland.com/Cpp/BugSrch.htm	<p>Borland C++ bug search URL</p>
www.pinpub.com/vcd/home.htm	<p>Visual C++ Developer Online. Online C++ magazine. About as useful as any other magazine, which means that if it has what your looking for, it's great. Otherwise, it doesn't matter much.</p>
www.vcdj.com/default.asp	<p>Visual C++ Developers Journal. Another online magazine.</p>
Walden.mo.net/~mikemac/clink.html	<p>Useful page for C++ links. Check it out.</p>

www.dinkumware.com/refcpp.html	Dinkum C/C++ Library Reference, in HTML format. Bears much resemblance to the VC++ standard C++ library HTML docs. This is probably because both were written by P.J. Plauger. So if you use a compiler other than Microsoft, this is a good place to get a good C++ library reference.
www.hal9k.com/cug/	The C/C++ User's Group. This group is sponsored by the C/C++ Users Journal.
www.research.att.com/~bs/homepage.html	Bjarne Stroustrup's (inventor of C++) page. Learn how to pronounce his name. Other stuff too.
www.cerfnet.com/~mpcline/c%2b%2b-faq-lite/	C++ FAQ page. Note that the web site is related to the book, C++ FAQs. Useful for honing your C++ knowledge if your in that middle ground between newbie and expert. Also, some good questions are answered here, such as "What do all the types of const mean?"
www.sgi.com/Technology/STL/ www.metabyte.com/~fbp/stl/effort.html	SGI STL. Read. Learn. Be impressed. Download.
www.experts-exchange.com/	Web site where you post questions to experts.
www.amazon.com www.computerliteracy.com www.barnesandnoble.com	Here's where you buy books. Amazon generally has the best prices (but not by much), and has a slightly better online bookstore than Barnes and Noble. Computer literacy is a great place to search for computer books and related things. Check out the Computer Literacy bookshop in Cupertino, Bay Area (a couple blocks from Apple Computer); it's a programmer's candy shop.

Gurus

Paul Pedriana (ppedriana@maxis.com) (paulp@ccnet.com)	As an attendee of the 1999 CGDC C++ Tutorial, you are entitled to some free advice from me through my personal email address. I am an expert on practical language issues, advanced syntax, debugging, optimization, and perhaps most importantly, architectural issues. I am not an expert on esoteric aspects of the C runtime library, compiler and build configuration, MFC, and console development. If you have a question about how efficient it really is to do this or that in C++ (or for that matter, C), I can probably give you a good answer. If you have a question about C++ syntax or usage, I can probably give you a good answer.
---	--